JOSES

Java Optimization Strategies for Embedded Systems

Workshop at ETAPS 2001

Dr. Uwe Assmann (ed.)
PELAB, Linköpings Universitet
S-581 83 Linköping, Sweden
Uwe.Assmann@ida.liu.se

April 1, 2001

Foreword

With the rising application of the Java programming language, the demand for powerful Java compilers is growing rapidly in all sectors of industry, in particular in the field of embedded systems. However, to produce efficient Java code for these systems many hard problems in compiler construction have to be solved.

The JOSES workshop (Java Optimization Strategies for Embedded Systems) called for innovative ideas in this area, solutions that meet the efficiency requirements of today's and tomorrow's embedded systems. Position papers were presented in 4 main areas, optimized memory management, static program analysis and optimizations, performance measurement and improvement, and run-time issues. The workshop presents an exciting overview of on the current work in the area. Several of the approaches presented have a great potential to optimize Java such that it meets the requirements of embedded systems very well.

The JOSES workshop took place as a satellite event of ETAPS 2001. My thanks go to Maura Cerioli, the workshop organizer, for her continuous help, the program committee, that took the work to review the papers, and finally, to all authors who made the event happen.

The JOSES program committee consisted of the following people: Prof. Dr. Uwe Aßmann, Linköpings Universitet, Sweden; Prof. Henk Sips, TU Delft, Netherlands; Prof. Reinhard Wilhelm, Universität des Saarlandes, Germany; Prof. Peter Fritzson, Linköpings Universitet, Sweden; Dr. Arthur Veen, ACE Associated Compiler Experts bv, Amsterdam, Netherlands; Dr. Pär Emanuelson, Ericsson SoftLab AB, Sweden.

On behalf of the JOSES program committee

Uwe Assmann

Session I

Memory Management for Java Programs on Embedded Systems

Analysis for Object Inlining in Java

Peeter Laud*

FR Informatik
Universität des Saarlandes
laud@cs.uni-sb.de

Abstract

Object-oriented languages allow to encapsulate the data; the unit of the encapsulation is an object. Programmers are advised to reflect the structure of the data while designing the layout of objects. This means that a lot of objects may be created. The semantics of Java dictates that such objects are stored in the heap and are accessed through references. Direct implementation of this would suffer from the overhead of memory management and pointer dereferencing. Object Inlining is an analysis to detect, whether some child objects could be stored together with their parent—the reference from the parent object to the child object would be replaced with the actual data of the child object. We present our analysis for Object Inlining, whose improvement over the previous ones is, that it can also detect, when a child object is replaced with a new one.

1 Introduction and Related Work

Java has a simple, uniform object model—all objects are accessed via references. This simplifies programming, as the programmer is faced with only one possible behaviour of objects. On the other hand, implementing such a model literally—each object is represented as a pointer to the heap, where the object's data resides—generates several kinds of overhead. The most explicit of them is the necessity to dereference a pointer each time an object is accessed. Also, the memory manager has to handle each object separately—the information that a group of related objects is always allocated and deallocated together is not available to it. Handling related objects together may also increase cache performance, if a profitable representation can be chosen for the *group.*

The object-oriented programming paradigm advises programmers to write the code in a way that increases the overhead described above. They are advised to distribute the data across several classes to reflect its structure.

In other languages, most notably in C++, the programmer can manually tune the representation of data in memory by declaring the fields of objects to be either objects or pointers to objects. In this way the program can be made

*Supported by the Esprit LTR project #28198 "JAVA and CoSy Technology for Embedded Systems" free of the overhead described before, but from the software design point of view, such non-uniform object model is of course inferior to the uniform one.

Our approach is to look for automatic possibilities of *object inlining*—determining, when the type of a field of an object can be changed from "pointer to object" to "object". The best known work in this area is probably that of Dolby and Chien, see [5] and the references it contains. Their approach is to find pairs of objects (r, e) together with a field f of r, where the only operation of storing a reference (i.e. an assignment of the form `x.f = y`), where e stands on the right-hand side, is $r.f = e$; the same operation of storing a reference also has to be the only one where $r.f$ stands on the left-hand side. This constraint ensures that the following program transformation does not change the aliasing information (the sharing patterns). We generalise their analysis by also considering the possibility that several objects $(e_1, \ldots, e_n)$ are inlined into the field f of an object r in succession. Still, the same requirement remains.

In a transformed program it is generally not statically known whether the field `f` in an expression `x.f` is a reference field or an inlined one. Also, the layout of the object pointed to by `x` is not known, hence the offset of the field `f` in this object is not known, either. Thus, if nothing is done about it, all field accesses must be done via a dynamic call of an accessor method. The analysis of Dolby and Chien makes sure that if the class (incl. layout) of the object pointed to by `x` is not known statically, then the field `f` is not inlined. Their cloning framework [7] appears to be well-suited for this. This ensures that dynamically dispatched accessor methods are not needed. Our analysis also handles the same issues (possibly more conservatively, as the cloning framework that we use is less powerful). Our approach differs from Dolby's and Chien's by clearly separating the issues of preserving sharing patterns (Sec. 3) and statically knowing, how to access fields (Sec. 6).

Our analysis proceeds as follows. We start with parts of the analyses that are described in Sec. 3, namely with those that do not require the results of type inference. They are described in the beginning of Sec. 3, in enumeration items 1 and 2. Cloning (Sec. 4), based on the results of these analyses, follows. Next are the type inference [4] and the rest of Sec. 3. This is followed by the data flow analysis described in Sec. 6 and another cloning, based on the results of analyses after the first cloning, except type inference, which is not a bit-vector analysis (and thus cannot be handled by our cloning framework). We finish by creating a constraint system over the values of the predicate "field `f` is inlined at objects created at the `new`-statement s" and the predicates described in Sec. 5 and 6. We feed this system to a generic constraint solver, together with the objective to inline as many fields as possible.

Our contributions are the semantic model of the heap and the analysis based on it, which are presented in Sec. 2 and 3. Also, we believe that our way of coinductively stating when the field accesses do not need dynamic dispatch, can be superior to the methods of Dolby and Chien, when one considers complex recursive data structures.

2 Semantics of the Memory

The memory at a certain program point consists of a set of variables **Var**, containing both global variables and local variables of all methods in the execution stack, and a heap $\mathcal{H}$. The heap is a set (with no further structure) of objects. There is also a set of fields **Fld**; each object is assumed to have all fields in it. There is a function

$$\mathbf{Ref} : \big(\mathbf{Var} \uplus (\mathcal{H} \times \mathbf{Fld})\big) \to \mathcal{H}_\perp,$$

where $\uplus$ denotes disjoint union and $\mathcal{H}_\perp := \mathcal{H} \uplus \{\perp\}$. This function describes, to which objects the variables and object fields point. *NULL* is denoted by $\perp$. We assume that the values of all fields and variables are references; we can overlook the atomic values for our purposes.

There is a predicate inl over $\mathcal{H} \times \mathbf{Fld}$, which says whether a particular field in a particular object is inlined or not. We require that if a field f of an object o is inlined (i.e. $\mathsf{inl}(o.f)$ holds) and $\mathbf{Ref}(o.f) \neq \perp$, then there exists no other field f' of an object o', such that $\mathbf{Ref}(o'.f') = \mathbf{Ref}(o.f)$.

We are going to define the *sharing pattern* of the memory, which must be preserved by program transformations. A pointer chain pc is either a variable in **Var** or a pointer chain followed by a field. Define a partial function $\mathcal{E}$ from pointer chains to objects in $\mathcal{H}$ as follows:

$$\begin{aligned} \mathcal{E}(v) &= \mathbf{Ref}(v), && \text{if } v \in \mathbf{Var} \\ \mathcal{E}(\mathsf{pc} \cdot f) &= \mathbf{Ref}(\mathcal{E}(\mathsf{pc}).f), && \text{otherwise.} \end{aligned}$$

We call the relation Ker $\mathcal{E}$ the sharing pattern of the memory $(\mathbf{Var}, \mathcal{H}, \mathbf{Ref})$ (this relation is actually the same as the *alias relation* of Deutsch [3]).

We assume to have the following statements in our programming language: `x = new(f1,...,fk)`, `x = y`, `x = y.f` (we call these statements “field loads”), `x.f = y` (we call these statements “field stores”), `x.f := y` (we call these statements “deep copies”) and statements for controlling control flow. The arguments of a `new`-statement are fields that will be inlined in the newly generated object. The semantics of `new`-statements and simple variable assignments is fully intuitive. The statement `x.f = y` denotes making the reference field `x.f` point to the object pointed to by `y`; it semantics is intuitively clear, too. The statement `x.f := y` denotes making the inlined field `x.f` reference the object pointed to by `y`. It works as follows: If the value of the field `f` of the object pointed to by `x` (i.e. the quantity $\mathbf{Ref}(\mathbf{Ref}(\texttt{x}).\texttt{f})$) was *NULL* before the assignment, then a new object is created (with the same inlined fields as $\mathbf{Ref}(\texttt{y})$) and `x.f` is set to reference it. Then, in any case, `x.f := y` makes a *deep copy* of the object pointed to by `y`—the value of `y.g` will be assigned to `x.f.g` for all $\texttt{g} \in \mathbf{Fld}$. This is recursive with respect to the inlined fields of `y` (which must be the same as the inlined fields of `x.f`).

3 Heap Analysis

We are given a program P that works with objects that have no inlined fields. We want to have a transformation that makes as many fields inlined as possible. More precisely, we are looking for a transformation that

- only changes the `new`-statements in the program (by marking some fields inlined) and turns some field stores to deep copies;

- is correct in a sense, that the resulting program P′ has the same sharing patterns as P at each program point, if we only consider live pointer chains.

Assume w.l.o.g. that the program P is such, that after each statement `x.f = y`, the variable `y` is dead. This can easily be ensured by following this statement by `y = x.f`.

The main constraint for inlined fields was, that no other field may point to the object that this field points to. The crucial point for the analysis is therefore finding out, the references of which objects may be stored in the fields of other objects several times. Also, for each field store we must know whether we change it to a deep copy or keep it as is. The analysis is as follows:

1. For each field store `x.f = y` find out, whether `y` may have flown from a field load. If this is the case, then this field store may not be turned to a deep copy. Also, the field `f` of objects that `x` may point at (and which are found by type inference [4]), cannot be inlined. This is done by a simple forward data flow analysis, which assigns to each variable at each program point either "not from a field load" or "may be from a field load".

2. For each field store `x.f = y` find out, whether `y` may have flown from a `new`-statement, and if this is the case, then find out whether `y` may have live aliases at this program point. If this is the case, then this field store cannot be turned to a deep copy, as this could change the fact, whether `x.f` and the alias of `y` point to the same object or not. This is done by a simple may-alias analysis that considers only variables, and not all pointer chains, see [6, Sec 4.2.3]. The results of this analysis have to be combined with the liveness analysis. Again, the field `f` of objects potentially pointed to by `x`, cannot be inlined.

3. For each field store `x.f = y` find out, whether the field `f` of all objects that `x` can point to, may be inlined. If this is not the case, then this field store cannot be turned to deep copy and the field `f` of none of the objects that `x` can point to, may be inlined. We have to iterate this marking of field stores that cannot be deep copies, and fields that cannot be inlined, until a fixpoint has been reached.

The analyses considered so far ensure that at a deep copy `x.f := y`, the right hand side has flown directly from a `new`-statement and that after this statement, the only way to access the object pointed to by `y`, is through `x.f`. We also have to ensure that the object **Ref**(**Ref**(`x`).`f`) is dead before a field store `x.f = y`, if we want to turn it into a deep copy. This amounts to determining, whether for each field load `w = z.f`

- `x` and `z` may be referring to the same object;
- `w` or variables, where it has flown, are alive at the statement `x.f = y`.

This is done by a forward data flow analysis that assigns to each variable at each program point a set of pairs ⟨`new`-statement, field⟩. For `w` at the statement `w = z.f`, the first components of the pairs would be the possible creation points of the object that is pointed to by `z`, and the second component of the pairs would be `f`.

We have thus found out, which objects cannot be put into inlined fields and which field stores can be deep copies and which ones cannot. From this information we can easily deduce, which fields of the objects at which creation points can be inlined fields and which cannot. The resulting transformation satisfies the conditions presented at the beginning of this section.

There exists a simple extension of the analysis and transformation, that does not preserve sharing patterns, but is obviously allowed. It concerns "constant objects"—objects, that are modified only at the beginning of their lifetime, i.e. that are initialised first and afterwards only read. The "beginning of lifetime" runs from the creation of the object until the object has been stored in a field of some other object. Obviously, one can create several copies of the same object later, because these copies cannot get "out of sync", as they are not modified at all. This increases inlining possibilities. Extra care has to be taken in transforming the comparisons of references that may point to constant objects, though. As they may point to different copies of the same object afterwards, an extra "identity" field should be added to such objects.

4 Cloning

Interprocedural data flow analyses use interprocedural control flow graph which, in itself, does not reflect that after the end of each method the control returns to the point where this method was called from. In general, this causes propagating the analysis information from some call-edge coming to the method, to a return-edge returning to some other call-site. This can cause the analysis to lose precision. Code reuse, promoted by the object-oriented programming paradigm, magnifies this problem.

A technique to avoid the propagation of information over invalid paths is *effect calculation* a.k.a. the functional approach to the interproc. DFA by Sharir and Pnueli. If the data flow analysis $\mathcal{A}$ assigns an element $\mathcal{A}(N)$ of an upper semilattice $\mathcal{L}$ to each program point N, then we transform the analysis in a way, that for each possible call context C of some method, the analysis assigns a value $\mathcal{A}_C(N)$ to the program point N inside that method. Additionally, we just let the call contexts to be all possible analysis results for the start node of the method, i.e. we let the set of analysis contexts be $\mathcal{L}$. In principle, this amounts to replacing $\mathcal{L}$ with $\mathcal{L} \to \mathcal{L}$.

Generally, effect calculation has a high price tag. It may greatly increase the complexity of the analysis or even make it nonterminating. For bit-vector analyses, the effect calculation can be made quite cheaply, however. In the case of a bit-vector analysis, the semilattice $\mathcal{L}$ has a quite small set of generators $\mathcal{B}$—the set of all vectors where a single bit is set. Also, the result of effect calculation $\mathcal{A} : \mathcal{L} \to \mathcal{L}$ is an upper semilattice homomorphism. This means that its values are uniquely determined by its values on some set of generators. Thus we only have to save the analysis results for all possible call contexts from the set $\mathcal{B}$ for doing the effect calculation. All analyses described in the previous section, except type inference, are bit-vector analyses.

Suppose that we use the analysis $\mathcal{A}$ for determining whether a particular transformation is valid. Also suppose that we have determined, that it is valid for some calling contexts $\mathcal{B}_{\text{good}}$ and invalid for contexts $\mathcal{B}_{\text{bad}}$. We can still do the transformation if we first create a clone of the current method and then redirect all calls to the method, that create bad calling contexts, to this clone.

In this way only calling contexts from $\mathcal{B}_{\text{good}}$ remain and the transformation can be done. If a call site generates both good and bad contexts, then we may attempt to clone the method that contains this call site.

5 Representing Objects in Memory

An object is represented in the memory of a computer as a sequence of fields. Thus, at `new`-statements, the layout of the object has to be fixed. For each field we have to fix its offset. If a field is inlined, then we also need to know how much room to leave for the objects that can be copied into it (the size of a reference field is fixed—it is the length of an address). For objects `o` and fields `f` we have an equation system to determine their sizes:

$$size(\mathtt{f}) = \begin{cases} \text{predefined}, & \mathtt{f} \text{ is reference (and also if it is atomic)} \\ \max\limits_{\mathtt{o} \in \mathit{Creations}(\mathtt{f})} size(\mathtt{o}), & \mathtt{f} \text{ is inlined} \end{cases}$$

$$size(\mathtt{o}) = \sum_{\mathtt{f} \in \mathsf{fields}(\mathtt{o})} size(\mathtt{f}) + \langle\text{administrative overhead}\rangle,$$

where $\mathsf{fields}(\mathtt{o}) \subseteq \mathbf{Fld}$ is the set of fields that are actually used in `o`; $\mathit{Creations}(\mathtt{f})$ is the set of objects (actually the set of creation points of objects), that `f` may point to; administrative overhead may contain information about object's runtime type, may include necessary attributes for memory management etc.

The constraint that follows directly from this system of equations is: No object may be inlined into its own field (not even indirectly). I.e. if $c_1, \ldots, c_n = c_0$ are creation points and $f_0, \ldots, f_{n-1}$ are fields and $c_{i+1} \in \mathit{Creations}(c_i.f_i)$ for all i between 0 and $(n-1)$, then at least one of the values $\mathsf{inl}(c_i.f_i)$ must be false.

Another constraint is caused by the type system of Java—each field of each class must have a fixed type. Thus, if the objects created at several different creation points may end up inlined into the same field, then these objects must be compatible enough to fit into the type system. The predicate $\mathsf{coinlinable}(c_1, c_2)$ describes whether objects created at the creation points c_1 and c_2 may be inlined into the same field.

$$\begin{aligned} \mathsf{coinlinable}(c_1, c_2) :\Leftrightarrow\ & \mathit{Class}(c_1) = \mathit{Class}(c_2) \wedge \\ & \forall \mathtt{f} \in \mathsf{fields}(c_1) : \Big(\big(\mathsf{inl}(c_1.\mathtt{f}) \Leftrightarrow \mathsf{inl}(c_2.\mathtt{f})\big) \wedge \big(\mathsf{inl}(c_1.\mathtt{f}) \Rightarrow \\ & \forall c_1' \in \mathit{Creations}(c_1.\mathtt{f})\, \forall c_2' \in \mathit{Creations}(c_2.\mathtt{f}) : \mathsf{coinlinable}(c_1', c_2')\big)\Big), \end{aligned} \tag{1}$$

where $\mathit{Class}(c)$ is the actual class that is named in the `new`-statement at the program point c. We see that $\mathsf{coinlinable}$ is defined recursively. To make this definition a correct one, we state that $\mathsf{coinlinable}$ is defined *coinductively*—it is the biggest relation satisfying (1) (for the given predicate inl).

We require that for all creation points c, for all fields $\mathtt{f} \in \mathsf{fields}(c)$ and all creation points $c_1, c_2 \in \mathit{Creations}(c.\mathtt{f})$ the formula $\mathsf{inl}(c.\mathtt{f}) \Rightarrow \mathsf{coinlinable}(c_1, c_2)$ holds.

statement	transfer function
`x = new`$_c$ `C`	$F_\bullet = F_\circ[\texttt{x} \mapsto \{c\}]$
`x = y`	$F_\bullet = F_\circ[\texttt{x} \mapsto F_\circ(\texttt{y})]$
`x = y.f`	$F_\bullet = F_\circ[\texttt{x} \mapsto \bigcup_{t \in F_\circ(\texttt{y})} \mathit{MakeTag}(t, \texttt{f})]$
`x.f = y`	$F_\bullet = F_\circ$

$$\mathit{MakeTag}(\langle c_1\texttt{f}_1 \cdots c_n\rangle, \texttt{f}_n) = \{\langle c_1\texttt{f}_1 \cdots c_n\texttt{f}_n c_{n+1}\rangle \mid c_{n+1} \in \mathit{Creations}(c_n.\texttt{f}_n)\}$$

Table 1: Analysis of accesses of inlined fields

6 Accessing the Fields of Objects Uniformly

We already mentioned in the introduction, that we do not want to use dynamically dispatched accessor methods to access the fields of objects. This requires the following:

- All objects that a reference variable may point to at a certain program point, must be compatible enough.
- If a reference variable at a certain program point *may* have been defined by a field load `x = y.f`, where `f` was an inlined field, then it *must* have been defined by such a field load. Our program transformation will be such, that a statement `x = y.f`, where `f` is an inlined field, will be changed to `x = y` and it will be statically remembered, that `x` actually points to the inlined field `f`. Such static remembering is impossible, if it could have been some other inlined field or no inlined field at all.

The required compatibility is the following: $\mathsf{copointable}(c_1, c_2, C)$ describes, whether objects created at the creation points c_1 and c_2 can be accessed through the same pointer with type "reference to an object of class C".

$$\begin{aligned}\mathsf{copointable}(c_1, c_2, C) :\Leftrightarrow\ & \mathit{Class}(c_1), \mathit{Class}(c_2) \le C \wedge \\ & \forall \texttt{f} \in \mathsf{fields}(C) : \Big((\mathsf{inl}(c_1, \texttt{f}) \Leftrightarrow \mathsf{inl}(c_2, \texttt{f})) \wedge \\ \forall c_1' \in \mathit{Creations}(c_1.\texttt{f})\, \forall c_2' \in \mathit{Creations}(c_2.\texttt{f}) &: \mathsf{copointable}(c_1', c_2', \mathit{Class}(C.\texttt{f}))\Big),\end{aligned}$$

which is again defined coinductively. For all creation points c, fields $\texttt{f} \in \mathsf{fields}(c)$ and creation points $c_1, c_2 \in \mathit{Creations}(c.\texttt{f})$ we require that the statement $\mathsf{copointable}(c_1, c_2, \mathit{Class}(c.\texttt{f}))$ holds.

For the second issue we record, from which fields of which objects the reference variables have been loaded. This part is similar to [5]. Let **Tag** be the set of strings $\langle c_1\texttt{f}_1 \cdots c_n\texttt{f}_n c_{n+1}\rangle$, where $n \ge 0$ and for each i between 1 and n we have $c_{i+1} \in \mathit{Creations}(c_i.\texttt{f}_i)$. The analysis associates each reference variable at each program point with a set of tags. Let $F_\circ$ and $F_\bullet$ denote the analysis result before and after some statement, respectively. $F_\circ$ and $F_\bullet$ are functions mapping variables to sets of tags. The transfer functions are depicted at Table 1.

The elements of **Tag** can be of any length, thus the described analysis may be nonterminating. The set **Tag** can be made finite, if we take into account that we are only interested in such tags $\langle c_1\texttt{f}_1 \cdots c_n\texttt{f}_n c_{n+1}\rangle$, where $\mathsf{inl}(c_i.\texttt{f}_i)$ may hold for all i between 1 and n. Thus, if we already know from the analysis presented in Sec. 3, that some $\mathsf{inl}(c_i.\mathtt{f}_i)$ does not hold, then we may identify this tag with $\langle c_{i+1}\mathtt{f}_{i+1}\cdots c_n\mathtt{f}_n c_{n+1}\rangle$. Also, we had the requirement that an object may not be inlined into a field of itself. Hence, if $c_i = c_{n+1}$, then this tag may be identified with $\langle c_{i+1}\mathtt{f}_{i+1}\cdots c_n\mathtt{f}_n c_{n+1}\rangle$. The last identification makes the set **Tag** effectively finite and the analysis terminating.

We require that all tags for some variable at some program point have the same length and contain the same fields in the same order. If this is not the case, then we have to not inline some fields.

We have to pick the "best" value for the predicate inl, that satisfies all the requirements presented above. Thus we have got an optimisation problem. The simplest definition of the objective function of the problem (i.e. the definition of "best") would just be the number of the fields that can be inlined. With help of some profiling information, one may be able to derive other, more advanced, objective functions. Still, we do not believe that the different maximal values of inl that satisfy all presented requirements, are very different, thus such simplistic objective function may be good enough.

Acknowledgements

We are thankful to Bruno Blanchet for helpful discussions. The described analysis has been implemented in the CoSy compiler construction framework [1]. Several of its component analyses have been implemented with the program analyser generator PAG [2].

References

[1] M. Alt, U. Aßmann, H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *5th Int. Conf. on Compiler Construction*, 1994.

[2] M. Alt, F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Static Analysis Symposium*, 1995.

[3] A. Deutsch. A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations. In *IEEE 1992 Int. Conf. on Computer Languages*, San Francisco, April 1992.

[4] H. Dewes, C. Probst. Static Method Call in Java. Submitted to *JOSES: Java Optimization Strategies for Embedded Systems*, 2001.

[5] J. Dolby, A. Chien. An Automatic Object Inlining Optimization and its Evaluation. In *Programming Language Design and Implementation*, 2000.

[6] W. Landi. Interprocedural Aliasing in the Presence of Pointers. PhD thesis, Rutgers University, 1992.

[7] J. Plevyak. Optimization of Object-Oriented and Concurrent Programs. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

Energy Optimization Using Object Co-Location in Java

S. Tomar, N. Vijaykrishnan, M. Kandemir, R. Shetty and M. J. Irwin
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802

ABSTRACT

With the paradigm shift in computer systems towards ubiquitous computing, energy, together with performance, has become an important parameter to measure efficiency. Java is increasingly becoming the programming language of choice for applications expected to run in embedded and mobile environments. Java's platform independence and security features serve the needs of these environments very well, which expect the same application to run in a variety of environments in a secure manner. The devices used in these environments, for example hand-held computers, have a limited battery life. The needs to increase the period between recharging and decrease the cooling costs provide the incentive to use energy as a performance parameter.
This paper presents object co-location, an optimization method for Java applications. Object co-location exploits the temporal locality in heap references, to achieve better cache performance. This reduces the cache miss rate of programs, and subsequent reduction in memory energy consumption is observed due to fewer main memory accesses.

1. INTRODUCTION

Computing is no longer limited to desktop computers and servers alone, but it is becoming more and more pervasive with the emergence of mobile and embedded systems. Devices are becoming smarter, and it is the result of a well integrated hardware-software interface. The trend of connecting all these myriad devices together calls for an environment which facilitates application development and ensures cross-platform delivery.

Java [1] is widely regarded as a platform for the seamless integration of these myriad devices. It is becoming one of the programming languages of choice for networked and embedded environments. Java's platform independence and security features make it suitable for these environments. Java Virtual Machine (JVM) [2], the cornerstone of Java technology has made it possible. Java programs are translated to a machine independent format called bytecodes, and these bytecodes are subsequently executed by an implementation of the JVM for that device/hardware. These bytecodes can be interpreted, compiled at runtime (called Just In Time (JIT) compilation) or implemented completely in hardware. The JVM specification provides only the semantics of these bytecodes, implementation of the runtime environment is left to the designer of the virtual machine. The JVM is also responsible for the efficient execution of Java programs.

The phrase *"efficient execution"* no longer refers to performance efficiency alone, but energy has also come to be included in it. There are two main reasons why energy consumption has become an important performance parameter. First, the energy dissipated as heat increases the packaging cost required for cooling the device. The increased cost is particularly important in low-cost end products in mobile environments. Saving energy in battery-operated embedded devices is also motivated by need to increase operational periods of devices between battery recharges. Many techniques have been proposed at the circuit and architectural level for energy optimizations. But software, which runs on these hardware also needs to be energy efficient, as it is the primary factor which determines the dynamic switching activity (and hence dynamic power dissipation).

In this paper, we present object co-location, an optimization scheme for Java applications. Using static memory profiles of Java programs, we present a methodology to exploit inherent temporal locality in heap references. Our results show that intelligent placement of objects on the heap can improve cache performance and energy. We also suggest a possible implementation of the scheme using a modified garbage collector component of the JVM.

2. RELATED WORK

Lately, work has been done to characterize and optimize memory system energy of Java applications[24, 6, 7]. A detailed study of memory energy in the JVM was done in [5, 24]. [6] provides an annotation based scheme for allocating arrays on the heap in a way that reduces energy. [7] also provides an energy profile of applications running on a hand-held device.

Improving the performance of dynamically allocated memory has been the subject of many research papers. Seidl and Zorn [12, 13] have investigated improving virtual memory performance by segregating objects based on lifetime, frequency of references and call sites. Following the generational hypothesis (most objects die young), and the fact that short-lived objects constitute a large portion of the objects allocated [12], a large number of objects can be allocated in a relatively small portion of the heap. This improves the spatial locality on the heap and leads to improved virtual memory performance.

Truong etal. [19] have suggested class field reorganization and instance interleaving as two data layout techniques for dynamically allocated data structures. Field reorganization groups more frequently referenced fields together in structure declaration so that they fit in the same cache line. Instance interleaving groups more frequently referenced fields of different instances of a data structure together. The argument behind this scheme is that identical fields in different instances of a data structure are often referenced together, and hence, they should remain in the cache together.

Chilimbi etal. [18] describes a generational garbage collection algorithm, in which objects with temporal locality are placed next to each other, so that they are likely to reside in the same cache block. Every load and store to heap data is profiled and based on this data, temporal relationships between objects are established. Objects with high temporal affinity are then placed next to each other on the heap during the next run of the garbage collector.

For Java, Chilimbi etal. [16] implemented a scheme called structure splitting. In this scheme, Java classes of instance size greater than a cache block are split into hot and cold portions depending upon which portion is used more frequently and vice versa, respectively.

Source code level changes have also been shown to improve performance [14, 15]. Liberal use of Java constructs leads to the creation of many objects and a high frequency of object-to-object copy operation [14]. These costly operations can be avoided by using techniques like object reuse, adequate object initialization and object and thread pooling [14].

3. MOTIVATION

Memory system can consume a large fraction of the total energy in Java applications and other embedded environments [24]. Hence, it is a good candidate for various hardware and software optimizations. For Java, heap is a very good candidate for memory optimizations, as a large fraction of all memory references go to the heap [4]. Further, it has been observed that most of the object references are confined to a small number of objects [17]. As a result, assigning random addresses to these objects can cause two problems in the cache. First, if two highly referenced objects are mapped to two addresses that conflict in the cache, then unless the cache is set associative, a significant number of conflict misses will occur. Second, the entries in a single cache line can be under utilized. Since the object addresses are assigned randomly, on a given cache line, only a small fraction of the line maybe assigned to a highly referenced object, while the rest of the line is essentially wasted because the objects that occupy it are infrequently referenced.

This observation leads us to conclude that assigning consecutive addresses to highly referenced objects can result in improvement in the cache miss rates. This is especially true for Java, because the average object size in Java applications is quite small, about 25-30 bytes [21]. The reason for this improvement is, that when a highly referenced object is fetched from the main memory on a cache miss, another highly referenced object is also fetched (because they are intelligently placed next to each other). Thus, there is good chance that when this object is referenced in future, it will be found in the cache and the penalty of a miss will be saved. The chances of this cache line being replaced are small, as the object adjacent to it is also frequently referenced. This reduction in cache miss rates translates into energy savings as accessing larger off-chip memories is much more expensive (in terms of energy) than accessing on-chip caches. In this paper, we experiment with an object co-location scheme, in which objects displaying temporal locality are co-located.

4. EXPERIMENTAL FRAMEWORK

In this section, we describe the framework we used for our experiments. We first give details of the JVM we used for running our Java applications. We then describe the benchmark applications and the cache simulator used for generating memory system information and profiling different components of the JVM. Our energy model is described after that.

4.1 Java Virtual Machine (JVM)

All our experiments were carried out using the Sun Labs Virtual Machine for Research (ExactVM (EVM)) [10], currently known as ResearchVM. EVM is designed to facilitate experimentation in memory management, especially soft real-time garbage collection. It is a high performance VM, which provides fast memory system, fast synchronization and a fast JIT compiler [10]. The EVM can execute Java programs in two modes: pure interpreter and an adaptive JIT mode. In the adaptive JIT mode, the EVM starts executing the application in the interpreter mode, and gathers *profiling information* regarding the runtime characteristics of the application. It then uses this information to dynamically compile certain methods. Specifically, methods with loops are compiled at the first invocation itself, while methods without loops are compiled when the invocation count reaches 15 [5].

4.2 Benchmarks

The benchmarks we used in our experiments are seven programs from the SPECjvm98 Benchmark Suite [3]. SPECjvm98 is an attempt to define an industry standard benchmark suite for Java programs. These programs are chosen by SPEC based on several criteria including high bytecode content, flat execution profile, repeatability, heap usage, and allocation rate [21]. Of the three input sizes of 1, 10, and 100, we have concentrated on the largest one (s100). Table 1 briefly describes each of the SPECjvm98 benchmarks. While the chosen applications are more likely to be executed in high-end (battery-operated) mobile devices like laptops, we believe the techniques presented in this work are also relevant to other applications typical of low-end mobile and embedded environments. We were constrained in our choice by the lack of a standard for embedded Java applications.

Cachesim5, which is a part of the Shade Tool Set [22], was used to obtain cache behavior statistics for benchmark applications. Shade simulates the execution of an application

Program	Description
db	Small data management program; performs database functions on a memory resident database
compress	Utility to compress/uncompress large files; makes five passes over the input
jack	A Java parser generator with lexical analysis; makes several passes over the same input
javac	The JDK 1.0.2 Java compiler
jess	Java expert system shell; based on NASA's CLIPS expert system
mpeg	MPEG-3 audio stream decoder
mtrt	Multi-threaded raytracer

Table 1: SPECjvm98 programs

and provides a programming interface that allows the user to collect arbitrary data while the application runs. Cachesim5 simulates a cache, traps each memory reference made by the application, and checks whether the reference will hit or miss in the cache. The information gathered at run time was fed to an analytical cache energy model [11] for our energy calculations. The model expresses cache energy in terms of the read and write port numbers, the number of registers, and several other relatively simple system and parameters for 0.35μ CMOS technology.

5. METHODOLOGY

Performance and energy results with objects co-located in memory were obtained in the following three steps:

Generating temporal information graph: In order to obtain information on how objects were temporally related, we generated a temporal information graph. This graph tries to relate all objects that are accessed within a specific window of references. To generate this graph, a program was written, which took object information generated through EVM, and memory reference trace as inputs. The program stepped through the memory trace of a benchmark, `n` references at a time (All experiments use $n = 100$). The objects in two adjacent windows (current and previous) of references are considered to have temporal locality and a node is assigned for each object in these windows and an edge is assigned between each pair of objects. To account for multiple references of an object within these adjacent windows, if an edge already exists the weight of the edge is incremented. This task is performed repeatedly for the subsequent windows until the entire memory trace is scanned. At the end of the entire scan of the memory trace, we had a temporal information graph, which depicted temporal affinity between objects. The higher the weight of the edge between two objects, the more closely (temporally) they were referenced throughout the program. The edges were then written to a file and sorted in decreasing order of weights. The generation of the temporal graph can be improved further by employing techniques such as sliding windows, and varying the size of the window, `n`. We are currently experimenting with alternate techniques for generating the temporal information graph.

Co-locating objects: The graph obtained in the first step was input to a program which co-located objects based on the information on temporal relationship contained in the graph. The program read edges of the graph in the decreasing order of weights.
Following algorithm (scheme `a`) was used by the program to place objects next to each other:

```
while(edges remain in candidate set) do
    select edge in the candidate set with the maximum
weight and remove it from candidate set;
if(neither object incident on the selected edge has
been relocated)
    relocate both objects;
else
    if(only one of the objects incident on the selected
edge has been relocated)
        relocate the other;
end while;
```

After determining the objects to relocate, the references belonging to the relocated objects were changed to reflect their new position. The trace with new references was then used as the input to the cache simulator to obtain cache statistics.

We also experimented with another allocation strategy. In this scheme, we identified the paths in the temporal information graphs that span all the nodes. The paths are selected to maximize the sum of the weights of edges across all the paths spanning the graph. All objects within the same path were then co-located.
The algorithm is presented below (we call it scheme `b`):

```
while(edges remain) do
    select edge in the candidate set with the maximum
weight;
if(neither node incident on the selected edge is found
in any existing paths)
    add this edge to a new path and remove the edge
from the candidate set;
else
    if((adding this edge does not cause a cycle) &&
      (does not cause any vertex to have degree > 2)
        add the edge to the appropriate path and remove
it from the candidate set;
    else
      remove the edge from the candidate set;
end while;
```

The paths selected were then used by our program to colocate objects. The traces obtained using the modified colocation algorithm were then input to the cache simulator to generate cache statistics. The purpose of experimenting with allocation scheme `b` was to take more temporal relationships into account (and not just between two objects). Since both the schemes `a` and `b` provided comparable results, the results are presented only for scheme `a`.

6. ENERGY SAVINGS WITH CO-LOCATION

Table 2 shows energy savings with co-location for seven SPECjvm98 benchmarks. For a 16K, direct mapped cache, average energy reduction of 21.36% (averaged over seven benchmarks) is observed. Figure 1 shows the absolute energy reduction obtained with co-location for benchmark `db`. Larger savings are obtained for bigger block sizes. It is expected, as, with a bigger block size, more objects can be brought into the cache at a time. These objects have been clustered together because they exhibit temporal locality, hence, soon other objects will also be referenced, and those references will not cause the additional penalty of a main memory access. These reduced main memory accesses result in energy improvements.

Figure 2 shows the effect of block size on energy savings obtained with co-location scheme. In going from a block size

Size	**Assoc**	`javac`	`comp`	`jack`	`db`	`jess`	`mpeg`	`mtrt`
16K	1	24.8	24.8	24.6	25.4	24.3	23.9	24.4
	2	21.5	21.4	21.6	21.6	21.5	20.8	21.0
	4	14.3	14.3	14.3	14.4	14.3	13.9	14.0
	8	39.6	39.6	39.6	39.6	39.6	39.5	39.6
32K	1	44.6	44.7	44.5	45.1	44.5	43.9	43.1
	2	30.8	20.8	21.2	20.9	20.7	20.4	19.8
	4	12.8	12.9	13.0	13.0	12.9	12.5	13.2
	8	7.6	6.7	7.6	7.7	7.7	7.4	7.6
64K	1	46.8	46.7	46.6	46.9	46.8	46.3	45.0
	2	48.8	48.9	48.9	48.8	48.9	48.4	48.7
	4	12.0	12.0	12.1	12.0	11.9	11.6	12.1
	8	6.7	6.7	6.8	6.7	6.7	6.5	6.9
16K	1	34.6	35.3	34.5	35.8	34.2	19.2	19.7
	2	15.1	15.2	15.2	15.2	15.1	14.3	15.5
	4	43.2	43.2	43.3	43.2	43.2	43.1	43.0
	8	8.2	8.2	8.3	8.2	8.2	8.4	8.3
32K	1	31.1	31.4	30.8	31.8	31.0	29.1	29.5
	2	14.4	14.6	14.7	14.6	14.3	13.8	14.0
	4	7.6	7.7	7.7	7.7	7.6	8.0	8.0
	8	6.5	6.5	6.5	6.6	6.6	6.7	6.4
64K	1	50.7	50.7	50.5	50.8	50.8	48.2	49.2
	2	16.3	16.4	16.3	16.3	16.3	15.4	16.4
	4	7.0	6.9	7.0	6.9	6.9	6.9	7.0
	8	5.7	5.8	5.8	5.7	5.7	5.5	5.8
16K	1	20.4	21.3	20.4	22.1	20.2	20.4	21.4
	2	40.9	40.9	41.0	41.0	40.9	43.5	43.8
	4	9.9	9.8	10.0	9.9	9.9	9.7	9.3
	8	9.5	9.5	9.5	9.5	9.5	9.2	9.2
32K	1	48.4	48.2	48.1	48.8	48.2	49.0	47.9
	2	9.7	9.8	9.9	9.9	9.7	8.0	8.3
	4	7.7	7.7	7.8	7.7	7.7	7.5	8.1
	8	6.4	6.4	6.4	6.4	6.4	6.3	6.0
64K	1	23.3	23.0	23.0	23.3	23.5	23.6	22.5
	2	10.5	10.5	10.5	10.6	10.5	15.6	10.8
	4	6.5	6.5	6.6	6.5	6.5	6.29	6.5
	8	45.3	45.2	45.2	45.2	45.2	45.3	45.3

Table 2: Percentage energy reduction with the co-location scheme as compared to similar configuration without co-location. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

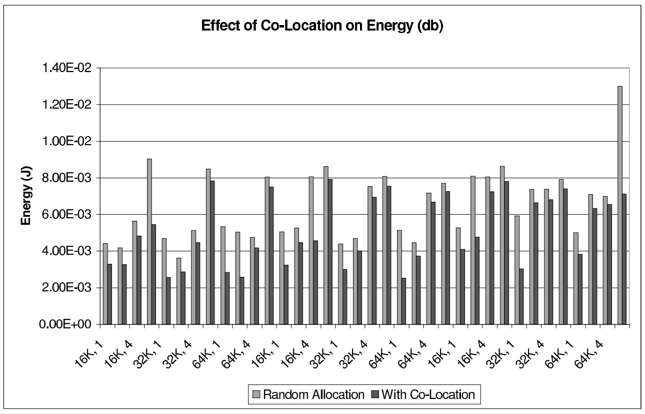


Figure 1: Absolute Energy values for `db`. Cache configuration are represented by the tuple (size in K, associativity). The results are presented for the different configurations for three block sizes 32, 64, and 128 bytes from left to right.

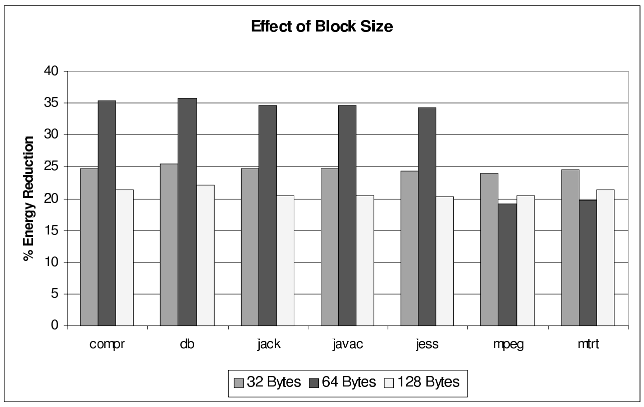


Figure 2: Energy savings for block sizes of 32, 64, and 128 bytes with the co-location scheme as compared to a similar configuration without colocation. Configuration is 16K, Direct-Mapped cache.

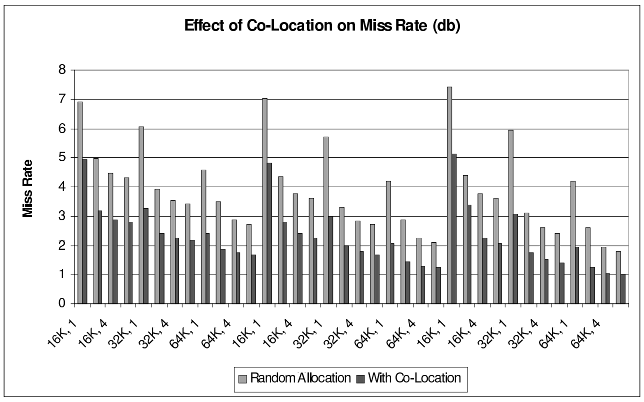


Figure 3: Miss rates with and without co-location for db. Cache configuration are represented by the tuple (size in K, associativity). The results are presented for the different configurations for three block sizes 32, 64, and 128 bytes from left to right.

of 64 bytes to 128 bytes, smaller improvements in energy are observed. The reason for this is lower reduction in the corresponding miss rates. Next subsection explains the reason why cache miss rates do not reduce as much as expected with co-location, in case of larger block sizes.

6.1 Cache Miss Rate Reduction

Reductions in cache miss rates obtained with co-location scheme are shown in Table 3. The results shows that significant improvements can be obtained from the co-location scheme in terms of cache performance. For a 16K, 2-way set associative cache, an average reduction of 35.4% is obtained. As can be observed, larger cache sizes of 32K and 64K get more benefits from the scheme.

Effect of block size: Figure 3 shows cache miss rates for the benchmark db. As would be expected, miss rates decrease with increasing associativity for a given cache size. Effect of block size can also be observed from this figure. Increasing the block size from 32 bytes to 64 bytes brings about larger benefits with co-location. This is because now, more objects which are temporally related can be fetched in one access. Because these objects are referenced relatively frequently with respect to each other, this results in larger benefits. This can be observed from Table 3 also.

Although the benefits from using larger block sizes can be seen with co-location, a general property of caches can be observed from the results: The reduction in cache miss rates for the random allocation case, and the co-location case individually, is smaller in case of block sizes of 64 and 128 bytes. This happens because the benefits obtained from bigger block size are somewhat offset by the fact that now a block takes up more memory in the cache, and there maybe more frequent replacements. It can be observed from figure 4 that the savings for block size 128 bytes are lower than those for block size 64 bytes.

Effect of associativity: Direct mapped caches gain more from the co-location scheme as compared with 2, 4, or 8-way set associative caches for both 32K and 64K caches

Size	Assoc	java	c comp	jack	db	jess	mpeg	mtrt
16K	1	27.9	27.8	27.7	28.7	27.3	26.8	27.5
	2	35.6	35.5	35.8	35.7	35.7	34.6	34.8
	4	35.1	35.0	35.2	35.4	35.1	34.8	34.7
	8	35.4	35.3	35.4	35.3	35.4	35.4	35.3
32K	1	45.7	45.8	45.4	46.4	45.4	42.7	43.3
	2	38.2	38.2	38.7	38.3	38.0	36.0	36.4
	4	36.4	36.6	36.7	36.7	36.6	37.0	37.3
	8	36.5	36.5	36.6	36.6	36.7	36.7	36.6
64K	1	47.3	47.0	46.8	47.3	47.3	42.3	43.1
	2	46.3	46.4	46.5	46.4	46.4	45.6	46.0
	4	39.1	39.1	39.4	39.1	38.9	39.1	39.5
	8	38.1	38.2	38.3	38.1	38.1	38.4	38.7
16K	1	29.6	30.6	29.4	31.3	29.0	29.7	30.5
	2	35.3	35.5	35.5	35.4	35.3	35.2	35.5
	4	35.9	36.0	36.1	36.0	35.8	35.3	35.1
	8	37.2	37.1	37.2	37.1	37.0	37.5	37.4
32K	1	47.0	47.3	46.7	47.9	46.8	44.6	45.1
	2	39.9	40.3	40.4	40.3	39.6	38.1	38.7
	4	37.8	38.0	38.0	38.0	37.8	39.0	39.3
	8	38.1	38.2	38.1	38.2	38.4	37.9	37.7
64K	1	51.1	51.0	50.5	51.3	51.1	46.7	47.4
	2	49.2	49.3	49.2	49.2	48.9	49.0	49.4
	4	42.0	41.9	42.2	41.9	41.7	41.9	42.3
	8	40.8	41.0	41.1	40.9	40.8	41.2	41.3
16K	1	28.9	29.9	28.8	31.1	28.5	28.7	30.1
	2	22.3	22.4	22.6	22.8	22.3	31.7	32.6
	4	39.8	39.7	40.1	40.0	39.7	37.9	38.0
	8	42.8	42.6	42.5	42.6	42.4	41.2	41.6
32K	1	47.8	47.6	47.3	48.6	47.6	46.6	47.0
	2	42.9	43.1	43.4	43.2	42.4	35.0	35.9
	4	41.3	41.4	41.5	41.5	41.3	42.4	42.7
	8	41.3	41.4	41.3	41.5	41.3	39.6	39.6
64K	1	53.5	52.8	52.8	53.4	53.6	51.3	51.5
	2	52.8	52.7	52.7	52.9	52.6	52.9	53.7
	4	44.9	44.9	45.1	44.9	44.6	44.5	45.0
	8	43.9	44.1	44.0	44.0	43.8	44.3	44.6

Table 3: Percentage reduction in miss rate due to co-location optimization. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

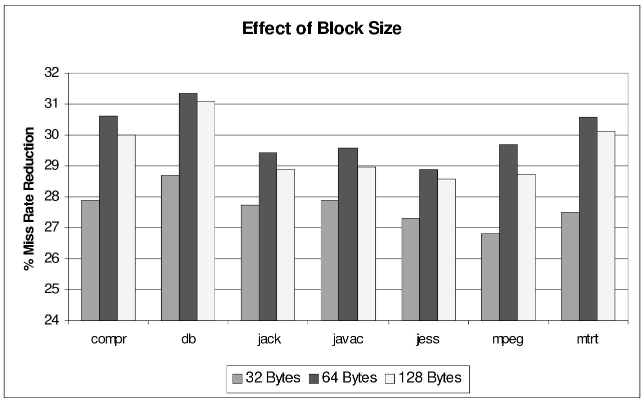


Figure 4: Effect of block size on cache miss rate reduction with co-location scheme. A 16K direct mapped cache is used.

in most cases. Higher associativity caches give lower miss rates even with random allocation. Direct mapped caches perform worse with random allocation because of the one-to-one mapping between main memory and cache addresses. Hence, direct mapped caches benefit more due to the conflict reduction after intelligent co-location of objects. This effect can be observed in Table 3.

7. DISCUSSION

Our results have shown that co-location can improve the performance of Java programs from performance and energy viewpoints alike. Our experiments were based on static memory profiles of SPECjvm98 benchmark applications. Therefore, we generated the temporal information graph based on this perfect knowledge about future. In practice, such information is not available while a program is running. Hence, we need some mechanism to obtain this information at runtime and provide it to the JVM. The JVM then needs to co-locate objects based on this information about memory references. In this section, we discuss how this information can be conveyed to the EVM, and present a possible solution.

Using garbage collector to co-locate objects: The garbage collector can be used to co-locate objects based on the temporal information made available to it. The EVM uses a compacting, generational garbage collector. If temporal information is available at garbage collection time, the collector can place candidate objects for co-location contiguously during the compaction phase.

We simulated the impact of such a runtime optimization by using our co-location method at each run of the garbage collector. A sentinel was inserted into the profiled memory reference trace every time the garbage collector was invoked by the virtual machine. This sentinel is used to create the temporal information graph whenever the garbage collector is invoked. Based on this temporal information, objects are co-located.

Table 5 shows the reduction in cache miss rate when this scheme we used. Corresponding reduction in energy values is shown in table 4. This simulated scheme performed as well as the scheme based on perfect knowledge. The applications in the SPECjvm benchmark suite are long running, and the garbage collector invocations are spaced apart by large number of memory (and heap) references. Hence, at each run of the collector, we have collected enough information about the past references to produce benefits from co-location.

8. CONCLUSION

Our experiments have shown that object co-location can significantly improve the cache miss rate of Java applications. We have gauged benefits from energy viewpoint also. In order to test a possible implementation of the scheme, we simulated co-location using the garbage collector to co-locate objects. Results obtained were comparable to the scheme based on perfect knowledge of all memory references. However, it must be noted that the creation of the temporal information graph and analysis will involve run time overheads. Our current study is on investigating low overhead techniques to translate the improved cache locality observed in this work to performance improvements in JVM implementations.

9. REFERENCES

[1] J. Gosling, B. Joy and G. Steele *The Java Language Specification*, Addison-Wesley, 1996.

[2] T. Lindholm and F. Yellin *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[3] Standard Performance Evaluation Corporation. SPECjvm98 Documentation, Release 1.0 August 1998.

[4] Jin-Soo Kim, Yarsun Hsu. Analyzing Memory Reference Traces of JAVA Programs. *Workshop on Workload Characterization*, October 1999.

[5] Anupama Murthy. Memory System Characterization of Java Applications. Master's Thesis, Dept. of Computer Science and Engineering, The Pennsylvania State University, May 2000.

[6] R. Athavale, N. Vijaykrishnan, M. Kandemir and M. J. Irwin. Influence of Array Allocation Mechanisms on Memory System Energy. To appear *International Parallel and Distributed Processing Symposium*, April 2001.

[7] J. Flinn, G. Back, J. Anderson, K. Farkas and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java Virtual Machine. Proceedings of *International Conference on Measurement and Modeling of Computer Systems.*

[8] L. Benini, A. Macii, E. Macii and M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application-Specific Memory Hierarchy Generation. *IEEE Design and Test of Computers*, Volume 17(2), pp. 74-85, April-June 2000.

[9] DineroIII cache simulator. http://www.cs.wisc.edu/~larus/warts.html#Tycho.

[10] The Sun Labs Virtual Machine for Research. http://www.sun.com/research/java-topics/.

Size	Assoc	javac	comp	jack	db	jess	mtrt
16K	1	25.52	12.96	24.93	26.01	25.35	11.03
	2	22.50	21.53	22.80	22.57	22.80	21.96
	4	14.95	14.96	15.02	14.98	15.14	14.52
	8	40.07	40.05	40.09	40.07	40.15	39.90
32K	1	45.69	45.73	44.79	45.36	45.54	44.26
	2	21.71	20.54	21.99	21.62	21.80	20.42
	4	13.48	13.49	13.68	13.53	13.61	13.60
	8	8.03	7.99	8.08	7.99	8.12	7.85
64K	1	47.51	47.00	46.98	47.19	47.40	45.70
	2	49.24	49.09	49.20	49.10	49.29	48.93
	4	12.44	12.35	12.50	12.37	12.40	12.33
	8	6.99	6.93	7.04	6.94	7.00	6.99
16K	1	36.58	17.94	35.38	37.35	36.20	5.95
	2	11.96	11.38	12.25	12.01	12.23	12.43
	4	44.46	44.27	44.48	44.42	44.53	44.19
	8	10.24	10.12	10.26	10.18	10.28	10.22
32K	1	32.70	33.39	31.36	32.62	32.42	31.69
	2	10.76	10.05	10.93	10.74	10.78	10.41
	4	45.25	45.27	45.29	45.23	45.26	45.42
	8	8.55	8.44	8.59	8.53	8.62	8.31
64K	1	64.50	64.35	63.91	64.24	64.30	63.70
	2	16.27	16.31	16.19	16.15	16.31	16.19
	4	44.67	44.63	44.68	44.61	44.65	44.64
	8	4.83	4.78	4.89	4.79	4.84	4.83
16K	1	21.95	6.28	20.78	22.96	21.71	3.23
	2	41.46	44.00	41.64	41.47	41.60	44.42
	4	10.60	10.22	10.68	10.58	10.78	9.91
	8	10.36	10.12	10.41	10.31	10.42	9.98
32K	1	49.62	49.40	48.55	49.32	49.43	49.82
	2	47.51	45.71	47.59	47.40	47.47	45.92
	4	8.28	8.23	8.41	8.31	8.35	8.61
	8	7.25	7.05	7.30	7.27	7.34	6.74
64K	1	21.68	21.00	21.12	21.38	21.62	21.44
	2	10.72	10.68	10.68	10.71	10.72	10.92
	4	6.96	6.97	7.06	6.96	6.98	6.93
	8	45.66	45.63	45.68	45.65	45.66	45.66

Table 4: Percentage energy reduction when colocation was performed at each run of the garbage collector. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

Size	Assoc	javac	comp	jack	db	jess	mtrt
16K	1	28.82	12.81	28.07	29.42	28.61	10.33
	2	37.37	35.76	37.74	37.41	37.75	36.34
	4	36.69	36.69	36.78	36.70	37.04	35.88
	8	37.06	36.99	37.11	37.05	37.36	36.40
32K	1	47.32	47.39	45.94	46.80	47.09	45.19
	2	39.79	37.66	40.16	39.55	39.90	37.52
	4	38.15	38.11	38.54	38.20	38.41	38.28
	8	38.18	38.02	38.28	37.98	38.45	37.55
64K	1	48.79	47.64	47.59	48.07	48.54	44.71
	2	47.44	47.02	47.31	47.05	47.58	46.57
	4	40.50	40.24	40.60	40.28	40.43	40.23
	8	39.43	39.13	39.59	39.24	39.50	39.30
16K	1	31.27	31.56	30.15	32.66	30.70	30.21
	2	37.56	36.11	38.01	37.56	37.98	37.85
	4	37.80	37.05	37.91	37.63	38.09	36.72
	8	39.18	38.68	39.23	38.93	39.32	38.93
32K	1	49.39	50.09	47.67	48.93	48.96	48.49
	2	41.84	39.78	42.08	41.62	41.81	40.49
	4	39.92	40.01	40.14	39.83	40.00	40.83
	8	40.03	39.46	40.13	39.86	40.28	38.98
64K	1	52.60	52.28	51.18	51.99	52.20	50.06
	2	50.38	50.42	50.08	49.99	50.42	50.25
	4	43.55	43.32	43.62	43.20	43.45	43.37
	8	42.53	43.25	42.77	42.35	42.62	42.37
16K	1	30.75	8.76	29.13	31.92	30.41	4.57
	2	23.63	32.65	24.38	23.80	24.28	34.27
	4	41.06	39.57	41.25	40.92	41.54	38.68
	8	44.23	43.18	44.37	43.99	44.36	42.77
32K	1	50.28	49.86	48.19	49.67	49.91	50.66
	2	44.78	36.61	45.12	44.29	44.62	37.64
	4	42.86	42.45	43.26	42.80	42.98	43.71
	8	42.55	41.32	42.71	42.58	42.87	40.11
64K	1	54.99	53.35	53.52	54.11	54.45	54.32
	2	53.94	53.66	53.68	53.46	54.06	54.29
	4	45.85	45.81	46.18	45.71	45.91	45.67
	8	45.37	44.97	45.54	45.22	45.28	45.31

Table 5: Percentage reduction in miss rate when colocation was performed at each run of the garbage collector. For the three cache sizes of 16K, 32K, and 64K, values are shown for block sizes 32, 64, and 128, in that order.

[11] V. Zyuban and P.Kogge. The Energy Complexity of Register Files. *International Symposium on Low Power Electronics and Design*, pages 305-310, 1998.

[12] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. Proceedings of *8th international conference on Architectural support for programming languages and operating systems*, October 1998.

[13] D. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. Proceedings of *SIGPLAN'93 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices 28(6), Albuquerque, NM, June 1993, ACM Press, pages 187-196.

[14] Reinhard Klemm. Practical Guidelines for Boosting Java Server Performance. Proceedings of *ACM 1999 conference on Java Grande.*

[15] Allan Heydon and Mark Najork. Performance limitations of the Java core libraries. Proceedings of *ACM 1999 conference on Java Grande.*

[16] Trishul M. Chilimbi, Bob Davidson and James R. Larus. Cache-conscious structure definition. Proceedings of *ACM SIGPLAN '99 conference on Programming language design and implementation*, May 1999.

[17] M. Siedl and B. Zorn. Low Cost Methods for Predicting Heap Object Behavior. Technical Report, Department of Computer Science, University of Colorado, Boulder, CO.

[18] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. Proceedings of *International symposium on Memory management*, October 1998.

[19] D. Truong, F. Bodin and A. Seznec. Improving Cache Behavior of Dynamically Allocated Data Structures. *International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[20] Bill Venners. Inside the Java Virtual Machine, Second Edition. McGraw-Hill, 1999.

[21] Sylvia Dieckmann and Urs Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Proceedings of *European Conference on Object-Oriented Programming*, June 1999.

[22] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Proceedings of *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128-137, May 1994.

[23] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, W. Ye and H. Kim. Energy-driven integrated hardware-software optimizations using SimplePower. Proceedings of *International Symposium on Computer Architecture*, June 2000.

[24] N. Vijaykrishnan et. al. Energy Behavior of Java Applications from the Memory Perspective. (To appear) Proceedings of *USENIX JVM Research and Technology Symposium*, April 2001.

Global Configuration of Cache Optimizations [⋆]

Rubino Geiß and Götz Lindenmaier

Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik, Universität Karlsruhe,
`{rubino|goetz}@ipd.info.uni-karlsruhe.de`

Abstract. Compiler constructors increasingly face a dilemma: Compute intensive applications that are written in Java to improve maintainability need to be executed on small embedded processors. Therefore we address cache optimizations for Java programs that target an embedded processor. We present a bytecode to native code compiler that performs these optimizations. It incorporates an analysis that finds Java arrays which can be subject to layout optimizations. A novel framework then determines a cache specific layout for these arrays. The framework can incorporate any layout optimizations that addresses a single loop. Based on proposals of such individual optimizations it decides on a dynamic layout for a whole procedure, i.e., if desirable the layout is changed between loops. The framework compares the efficiencies of the proposals and also considers the cost of establishing the layout. With preliminary experiments we show the potential of our framework.

1 Introduction

Object oriented languages are designed to support writing structured programs that are easy to maintain and to improve programmer efficiency. Many embedded applications are developed for different systems that perform similar tasks. Nowadays these systems reach sizes that make careful software design an important factor in product design. This development asks for tools and languages supporting careful software design. Unfortunately object oriented languages, and in particular Java, do not achieve the performance necessary on typically small embedded systems. Therefore the EU project JOSES explores compiling Java to native code and optimizing it for embedded systems.

Many embedded applications deal with large amounts of data, as, e.g., any applications that record or display video and audio data. Therefore recent embedded processors use caches to speed up processor performance. Due to the size constraints of these processors the caches are less powerful than those in multipurpose processors and require programs specially tailored for them. This can be achieved by programming under consideration of the memory architecture or by

[⋆] This work is supported by Esprit LTR Project #28198 "JAVA and CoSy Technology for Embedded Systems" (JOSES). Any opinions, findings, and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect the views of the sponsors.

support of the compiler. Java does not allow to consider memory aspects during programming as writing portable and maintainable software prohibits processor specific coding. Therefore Java compilers for embedded systems need automatic cache optimizations.

Cache optimizations employed in a compiler have two basic handles to improve a program: code transformations and data layout transformations. Code transformations typically transform compute intensive loops. This changes the order individual iterations are executed in. Java defines very restrictive exception semantics: Code, and in particular loop iterations, must be executed in the order of the source code if exceptions can occur in this code. I.e., loop transformations for Java require to prove that the loop can not cause any exceptions. Layout transformations do not raise this problem as they do not affect the execution order. The layout of objects and especially arrays in memory is not defined by the Java semantics – any changes are legal. A layout change merely affects the addresses used during program execution.

This paper presents an approach to optimize the layout of arrays in Java programs. We automatically find dynamic allocated arrays that can be allocated statically so that the compiler can optimize their layout. We propose a framework that allows to choose from and combine a set of different layout optimizations for individual loops. Further our framework implements an algorithm that chooses the layout under procedure global aspects. Typically layout transformations are performed for the scope of an individual loop nest or independent of the actual access pattern [BCcRJ$^+$94,RT98,PNDN97], often specifying a static layout for a procedure with a single loop nest by adjusting declarations of local variables. Our algorithm in contradiction chooses a dynamic layout, i.e., if desirable the layout is changed during runtime. In contradiction to other work we further consider the cost of changing the layout [LRW91].

The next section explains the compiler environment our optimization is placed in. Section 3 details how the optimization decides which arrays can be allocated statically. We explain our cache optimization in detail in Section 4. In Section 5 we present and discuss first experimental results and in Section 6 we delimit our approach from related work. Finally we conclude.

2 Cache optimizations in JOSES

Our cache optimizations are integrated in the JOSES optimizing compiler [Vee01], i.e., they operate on its intermediate representation. This compiler is based on the compiler construction framework CoSy [AAvS94] and translates Java to executable binaries. It employs three different intermediate representations (IR): OMIR, a traditional IR incorporating object oriented constructs; SMIR, an IR in SSA form developed at University of Karlsruhe [TLB99]; and finally CCMIR, a low IR with the object oriented constructs removed.

The optimization is separated into three phases. The first phase finds arrays that can be allocated on the stack. For each such array it changes the type represented in the IR, the local variable declarations and all accesses to the array. The second phase analyzes the control flow graph of each procedure for optimizable loops and gathers all those arrays whose layout can be controlled by the compiler and can be subject to certain optimizations. The third phase is the actual optimization system we call OptiCache. This system divides into several further steps internally. It is explained in detail in Section 4.

3 Analysis of Java Arrays

This section details the analysis of phase one as introduced in Section 2. The goal of this analysis is to find Java array objects that are actually used as local array variables of a procedure. An inter-procedural heap analysis that is under construction [Lie00] will assist this analysis.

To optimize the layout of arrays during compile time it is necessary that the compiler has precise information about them. This is not possible with arrays as used in Java. Dynamic allocation deprives the compiler from controlling the base address and size of an array. The treatment of multidimensional arrays as cascading arrays of arrays obstructs controlling the layout of the dimensions.

```
FOR (all calls to array allocation function)
  Evaluate size expression;
  IF (size expression not constant) NEXT call;
  FOR (all users of pointer returned by call)
    IF (user is Store) NEXT call;
    IF (user is Call) NEXT call;
    IF (user loads length fields) remember node in list1;
    IF (user selects array field) remember node in list2;
  add local array variable;
  remove call to array allocation;
  FOR (all members in list1)
    replace by constant length;
  FOR (all members in list2)
    replace by access to local array;
```

Fig. 1. Basic algorithm to find static arrays.

The analysis takes advantage of SMIR. SMIR represents a procedure as a data flow graph. All local variables are data flow edges. Any side effects are represented by explicit load and store nodes in this graph. To sequentialize these operations the memory that is used by the program is modeled as a data flow value itself. A data flow edge that represents a local array must start at a procedure call node that calls the compiler known array allocation function and returns a reference to the array. The analysis must assure that such edges only end at nodes that access individual array elements. In particular they may not end at store or call nodes. This would mean that the reference to the array can be copied or passed out of the procedure. A stronger analysis would try to find out whether such a case actually produces an alias. This would find more arrays and thereby amplify the effect of the optimization.

The call node for the compiler known array allocation function in the SMIR graph has an incoming edge that represents the expression that computes the size of the array to allocate. A strong procedure global constant propagation and constant expression evaluation algorithm tries to evaluate this expression to a constant. If it succeeds, this array can be transformed to a static array in the procedures stack frame.

The compiler creates a new local variable for the array of the known fixed size and removes the call to the array allocation. Now all accesses to the length field of the array need to be replaced by the constant length. As a side effect this turns loops that iterate over the array by loading the upper loop bound from the array object into loops with constant loop bounds making them easier to optimize. Further the array accesses need to be replaced by accesses to the array on the stack. As a side effect this removes one indirection in accessing the array fields. This algorithm is summarized by Figure 1. Due to the structure of SMIR the cost of this analysis is proportional to the number of nodes actually changed in the IR. Only the cost of the search for calls to the allocation is proportional to the number of nodes in a procedure.

```
int [] a;                             int a[14];
int b = 7;                            int b = 7;
int i;                                int i;
a = new int [b*2];
a[3] = b;                             a[3] = b;
for (i=0; i<a.length i++) {...}       for(i=0; i<14; i++) {...}
```

Fig. 2. Java code and transformed code represented in C.

Figure 2 shows the effect of the transformation with an example. The left program fragment in Java shows an array that is allocated with a size computed from constants. Then an array field is accessed. This access involves two offset computations (stack frame pointer plus offset of a and array base plus array index) and a dereference (load `a` to get array base). The access of the length field, which typically is array element -1, requires the same indirection. The right program fragment only exists in the internal representation of the compiler, but here this is expressed in C. The size expression is evaluated and the array is allocated statically on the stack[1]. The access of the array element does not require a dereference any more (This can not be seen explicitly in this code). Instead of the access to the length field a constant is used. The loop now has constant bounds.

[1] Here we actually change the semantics of the program. A possible out of memory exception would not occur any more, instead we could get a stack overflow.

4 The Cache Optimization

This section details the analysis of phases two and three as introduced in Section 2. The herein described optimizer uses a problem representation that is separate to the IR. This is built up by analyzing CCMIR in phase 2 and enriched in each step of the optimizer with new pieces of information. Finally it represents a layout for the arrays in a procedure that is a dynamic optimal configuration out of the proposals.

4.1 Gather Information

In the second phase we gather all needed information from the IR and check the optimize-ability of the procedure.

First we find all loops in the IR and build a data structure to represent them. This is done by a module that comes with the CoSy framework[2]. Based on this we to judge whether a loop nest meets the criteria necessary for optimization. If the procedure does not contain any loop optimization of this procedure is futile. If there are loops we have to check whether the loops are strong-for-loops. A loop is a strong-for-loops if and only if it is cleanly nested, has no procedure calls or conditions in its body and the loop bounds are compile time constants.

Furthermore there are things necessary to know about the arrays accessed in those loops. All those arrays have to be local. With some more precaution we can handle globally defined arrays, but this is not done here. The optimization can tolerate the existence of other arrays, but it can not control the effects between the optimized and the other arrays. Therefore we decided to optimize only procedures where we can control all arrays. All accesses to the arrays have to be affine in the iteration variables. This is a convenient restriction that makes it easier to program an effective cache simulator as detailed later. The size of the arrays have to be known at compile time. Finally we have to assure that none of the arrays considered for optimization are passed to a called procedure.

All information talked about in this subsection resembles the initial problem representation.

4.2 OptiCache

The OptiCache System [Gei00] itself consists of three steps: First, we dispense some layout proposals according to the problem representation. Then we choose one of those layouts per loop nest, and finally we implement this configuration into the IR.

Layout Proposal To propose a data layout we use simple heuristics. It would be a bad idea to do really clever things here, because most of them will not be chosen. It is much smarter to get plenty proposals such that there is a chance

[2] The loopmarker engine in the CoSy framework.

they will fit together without the need (and extra work) of a dynamic layout change.

We use the following three sources of proposals:

- No padding at all.
- Padding with 1·cache line size, 2·cache line size, 3·cache line size.
- A modified version of the InterPadLite-Algorithm introduced in [RT98].

The framework can easily be extended by further heuristics as, e.g., transposition or merging of arrays.

Choice of Layout This phase is the core of the optimization. We do some very computational intensive tasks to get a rather exact cost model as well as to find an optimal choice based on that. To get this done we introduce a graph representation of our problem. With this view of the problem we can simplify its structure that far that we are able to use standard operations research tools to solve it.

We want to describe the influence of different data layouts in different loop nests in terms of cache misses. Therefore we have to keep two things in mind: First, the execution of a loop nest in respect to a certain data layout of the array accessed in it. Second, the extra work (and cache misses) needed to change the data layout dynamically. The second task is easy to do as we define the template for the dynamic array layout change ourselves. We only have to provide a parametric cost function in terms of array size, dimension etc. The first task is not that easy, because we have to predict the cache behavior. Systematic approaches to estimate the amount of misses restrict the optimizable programs as certain constraints need to be met. Further these tend not to be exact, i.e., do not permit an optimal choice out of the proposals. Others as the Cache Miss Equations proposed in [GMM97] are very hard to implement and computational intensive. Therefore we chose to simulate the behavior of the loop nest in terms of cache accesses. To control the runtime of the simulation we plan to employ a randomized simulation. By simulating only parts of the iteration space of a loop nest we can scale runtime and preciseness of the simulation.

To use a standard binary program solver (an operation research tool like the "lp_solve" system [Sch96]), we transform our problem representation into a graph-like structure and combine the different costs into an uniform measure.

Layout Implementation Finally we have to implement the chosen global configuration of layouts in the IR. This is done by replacing the access to all optimized arrays by an access to a pointer variable of appropriate type. Before every loop nest we let these pointers point to the actual data location in memory. We have to completely control the layout of this data memory on the stack; it may not be rearranged by the backend. If there is a layout change between two loop nests, we insert a suitable loop in between.

5 First Experimental Results

In this Section we describe first experiments with our optimization and the JOSES compiler.

The test programs are compiled with the JOSES compiler. As the backend for the TriMedia is not yet appliable we use a C outlet. Then we compile the output code with the TriMedia C compiler and include a rudimentary runtime system. We execute the result on the TriMedia (TM 1100, 125MHz). To measure the performance we instrument the intermediate C code.

```
float [] a = new float [131072]
int   [] b = new int   [131072]
for (i = 0; i < 131072; i++)                /* nest 1 */
    a[i], b[i]
for (i = 0; i < 131072 - 3584; i++)         /* nest 2 */
    a[i+   0], a[i+ 512], a[i+1024], a[i+1536]
    a[i+2048], a[i+2560], a[i+3072], a[i+3584]
    b[i+   0], b[i+ 512], b[i+1024], b[i+1536]
    b[i+2048], b[i+2560], b[i+3072], b[i+3584]
for (i = 0; i < 131072 - 3600; i++)         /* nest 4 */
    a[i+  16], a[i+ 528], a[i+1040], a[i+1552]
    a[i+2064], a[i+2576], a[i+3088], a[i+3500]
    b[i+   0], b[i+ 512], b[i+1024], b[i+1536]
    b[i+2048], b[i+2560], b[i+3072], b[i+3584]
for (j = 32; j < 33*16; j+=16)              /* nest 4 */
    for (i = 0; i < 131072 - 4096; i++)
        a[i+j+   0], a[i+j+ 512], a[i+j+1024], a[i+j+1536]
        a[i+j+2048], a[i+j+2560], a[i+j+3072], a[i+j+3584]
        b[i+     0], b[i+   512], b[i+  1024], b[i+  1536]
        b[i+  2048], b[i+  2560], b[i+  3072], b[i+  3584]
```

Fig. 3. Example program.

We equipped our compiler with a set of flags to influence the optimization. The following list explains the settings of flags we use for the experiments.

jvm Bytecode executed on a Java Virtual Machine (kaffe). This code is executed on a 400MHz Pentium II as we do not possess a jvm for TriMedia.
java Java code translated without any optimizations, i.e., it contains dynamic allocated arrays.
loc arr Arrays found by analysis transformed from dynamic allocation to allocation on stack. No cache optimization.
static A static padding for the analyzed arrays.
dyn A dynamic padding for each loop.
glob Global configuration of dynamic padding.

Figure 3 shows a first example program with a set of loops containing numerous array accesses. The offsets used to access the two arrays are chosen so that a single layout can not suffice for all loops.

Table 1 shows execution numbers. It lists the total execution time (row "total") in milliseconds for the 6 different versions of the program. Rows "1" to "4" give the execution times of the four loops of the example program executed on the TriMedia. Further the table lists the time spent to fix the dynamic layout (row "pad").

The code executed on the jvm is about as fast as the native code. As the jvm code is executed on a far faster machine this states that the native code is actually faster. The program with arrays as local variables is slower than the program with dynamic allocated arrays. Dynamic allocation chooses base addresses of arrays randomly. This corresponds to a random pad. The numbers for the individual loops show that the random pad generates a few misses for the third loop and performs the worst for the fourth loop. The static layout manages to reduce the misses for the fourth loop, but must choose a layout that is suboptimal for the third loop. Only the dynamic layout manages to find layouts that perform well for all loops. It chooses three layouts, one for the first, one for the second and third, and another for the last loop. The global configuration now decides for a single layout for the first three loops and eliminates one of the loops needed to change the padding. This version also is the fastest. The table further shows that the cost of the loops increases almost linearly with the number of accesses in the loop. Further the gain by reducing conflict misses (about 400ms) outweighs the cost of the layout change (7 ms) by far.

program	loop	jvm	java	loc arr	static	dyn	glob
demo	1		9	11	11	11	11
	2		98	512	98	98	98
	3		109	97	482	95	96
	4		3532	3523	3113	3113	3113
	pad		–	–	–	3*7	2*7
	total	3560	3752	4147	3710	3346	3338

Table 1. Performance numbers (execution times in milliseconds)

As this is a constructed example this can only be a first step in testing our optimization. But these numbers show the potential of cache optimizations for data intensive Java programs.

6 Related Work

Several researchers concentrate on optimizing the layout for individual loops by padding. Recent work on Padding is performed by [BCcRJ$^+$94], [RT98] and [PNDN97]. Their heuristics propose paddings between arrays or between rows of multidimensional arrays. The algorithms all try several pad sizes until they find a suitable one, but they differ in the way of evaluating the use of a certain pad size. This evaluation depends on the side conditions (access expression complexity, loop structure). The padding is determined for a single loop or access pattern and is performed statically by changing the variable declarations of Fortran programs.

To us no approach is known that effectively uses dynamic layouts. Copying in tiled loops is the only work in this direction. [TGJ93] improve work first proposed by [LRW91] on copying to reduce conflicts in tiled loops. As loop tiling is a code transformation it is not applicable in our context. [PNDN97] discuss copying arrays passed to library routines to establish a padding suitable for the computations in the routine, but do not propose a technique that decides whether this copying will pay off.

Cierniak and Li ([CL98]) follow a similar approach as the JOSES project. They build a compiler reading bytecodes and performing target specific cache optimizations. They also have to recover nesting structures of loops, array access expressions and allocation statements. They implement array transposition for arrays allocated with the `multinewarray` bytecode. In [CL95] they give an algorithm to combine loop and layout transposition for several loops for Fortran. Their algorithm finds a static layout for the arrays that allows to optimize a maximal number of loop nests with loop transposition. They mention that several accesses to an array might cause conflicts within the array but do not give a solution. They do not consider to change the layout during runtime. [CL98] does not detail how they implement this optimization in their Java compiler, but as they argue that they can not perform loop transformations for java they obviously skip these. The restriction to transposition of arrays allocated with `multinewarray` simplifies the optimization: they can keep the dynamic allocation of the array.

7 Conclusion

With this work we present a unique framework to unify layout transformations. We show how to apply these to Java programs, and that it is feasible to apply them to embedded systems. Our framework considers dynamic layout changes as well as their cost. It is implemented in a real compiler that targets an embedded processor.

Our work is not yet complete, neither the optimization nor the compiler. It is necessary to finish this work and profoundly evaluate the optimization. We plan to add other layout transformations to our framework as transposition or merging. Further we will explore how to reduce the cost of evaluating and comparing the benefit of individual optimizations.

References

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy compiler phase embedding with the CoSy compiler model. In *Proceedings of the 5th In-*

ternational Conference on Compiler Construction, volume 786 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, April 1994.

[BCcRJ$^+$94] D. F. Bacon, J. Chow, Dz ching R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON '94*, pages 270–282, Toronto, Canada, November 1994.

[CL95] M. Cierniak and W. Li. Unifying data and control transformations for distriubed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 205–217, San Diego, CA, USA, June 1995.

[CL98] Michal Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 1998. to appear.

[Gei00] Rubino R. Geiß. OptiCache – Schleifenübergreifende Cacheoptimierung. Master's thesis, Dept. of Computer Science, University of Karlsruhe (TH), September 2000.

[GMM97] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.

[Lie00] Florian Liekweg. Design of Heap Analysis and Static Garbage Collection. Technical Report 8505, The JOSES Consortium, May 2000.

[LRW91] M. S. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, USA, April 1991.

[PNDN97] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and A. Nicolau. Improving cache performance through tiling and data alignment. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1253 of *Lecture Notes in Computer Science*, pages 167–185. Springer-Verlag, Paderborn, Germany, 1997.

[RT98] Gabriel Rivera and Chau Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[Sch96] Hartmut Schwab. Documentation of lp_solve. Technical report, 1996. Contained in lp_solve package available from ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.

[TGJ93] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, USA, November 1993.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-44, Dept. of Computer Science, University of Karlsruhe (TH), December 1999.

[Vee01] Arthur Veen. The JOSES Project: Compiling Java for Embedded Systems. In *Proceeding of JOSES: Java Optimization Strategies for Embedded Systems*, Genova, Italy, April 2001.

Hard Real-Time Reference Counting without External Fragmentation*

Tobias Ritzau

Linköping University, 581 83 Linköping, Sweden
tobri@ida.liu.se

Abstract. Using automatic memory management eliminates many programming errors that are both hard to find and to correct. Automatic memory management has been used frequently in functional and object oriented languages. These languages have rarely been used in hard real-time systems in the past. However, Java^{TM} has made the hard real-time community interested because of it robustness and platform independence. Introducing Java^{TM} in the hard real-time domain causes many problems. One of them is how to adapt the automatic memory manager to be fully predictable in both execution time and memory usage. This paper proposes such a technique.

1 Introduction

Automatic memory management (garbage collection or GC) is used to reclaim memory that is no longer used by the application. This greatly reduces the risk of memory related errors in programs. Memory related errors are often very hard to find and correct, since they may be located at places other then where they appear, e.g. deallocating the same memory region twice may not cause a problem until the region is reused.

Reference counting [7] is known to have many disadvantages, and many will not use it because of those. However, reference counting also has numerous advantages, especially for use in embedded and distributed systems.

The idea of reference counting is simple: count the number of references to every object and recycle the object if its reference count becomes zero. A reference counting memory handler consists of two main operations: *increment* and *decrement* reference counters. The decrement operation also handles deallocation when a reference counter becomes zero. These operations can be implemented as shown in Figure 1.

The advantages of reference counting are its *simplicity*, *fine grainedness* (only a few instructions need locking in multi-threaded systems), it *does not use separate garbage collection code to reclaim memory* (all work is done by the increment and decrement operations), and that data is *not moved* back and forth between different memory regions.

* Supported by the ESPRIT LTR project #28198 "JAVA and CoSy Technology for Embedded Systems".

```
algorithm incrc(obj)
    if obj $\neq$ null then
        obj.rc $\leftarrow$ obj.rc + 1
    end if
end

algorithm decrc(obj)
    if obj $\neq$ null then
        obj.rc $\leftarrow$ obj.rc - 1
        if obj.rc = 0 then
            foreach c in obj.children() do
                decrc(c)
            end loop
            free(obj)
        end if
    end if
end

algorithm assign(lhs,rhs)
    incrrc(rhs)
    decrrc(lhs)
    lhs $\leftarrow$ rhs
end
```

Fig. 1. Standard reference counting

Having a simple technique makes reference counting more feasible in safety critical systems, since it is easier to implement and to prove correct. Because of its fine grainedness, reference counting is suitable for multi-threading, especially for uniprocessor systems where locking is cheap. Since all updates are performed by the increment and decrement operations, there is no need to execute garbage collection code when allocating memory or at any other time other than when performing reference count updates. This simplifies worst-case execution time analysis. Reference counting keeps data in its fixed memory locations, in despite of copying garbage collection techniques. Keeping data in fixed locations simplifies interfacing to other systems and languages. Copying data also requires more memory, it takes time and makes synchronization between the garbage collector and the program harder.

However, there are also some disadvantages. *Cascading deallocation* occurs when the last reference to a large data structure is removed. This may cause long interruptions that may be hard to predict. Since memory is not compacted, *fragmentation* may be an issue. In most systems fragmentation causes no problem, but it may in a safety critical system. Reference counting is often considered to be *slow*, and the basic technique is. The inability of reclaiming cyclic data structures is often considered the major drawback of reference counting.

Cascading deallocation can be eliminated using a technique described by Weizenbaum [16]. The decrement operation is changed to put the object into a list instead of decrementing its children and deallocating it. If the system runs out of memory when allocating an object, the list is processed and dead memory is reclaimed. This solves the problem of cascading deallocation, but garbage collection code needs to be executed when allocating objects. The technique is designed for systems where all objects are of the same size, but can easily be adapted to suit other systems as well.

Splitting objects into blocks of equal size can be used to handle external fragmentation. This approach is further discussed below.

Several techniques to reduce the execution time overhead of reference counting have been proposed. All aim to reduce the number of reference count updates either by not counting all references [8, 1] or by statically finding redundant reference count updates [2].

Cyclic data structures can be reclaimed using several techniques. These include weak pointers [5], partial mark-sweep [6] and reference counting cyclic data structures instead of their components [3]. Partial mark-sweep algorithms require no manual administration, but the others do. This can cause problems since people are known to make mistakes, so they should be used with caution in safety critical systems. Partial mark-sweep techniques are feasible in interactive systems, but have not been proven to work in hard real-time systems.

In this paper everything that is allocated on the heap is called an *object*. A *dead object* is an object that will not be used by the program anymore. The *children* of an object are the objects that are referred by the object.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 proposes a technique to eliminate external fragmentation in a reference counting system. Section 4 discusses the worst-case execution time of the operations in the proposal. In Section 5 a prototype implementation is presented. Section 6 presents an evaluation and Section 7 presents the conclusion and future work.

2 Related Work

We classify real-time systems into three categories: interactive, soft real-time and hard real-time. *Interactive systems* are any systems that interact with the external world, e.g. X-Windows. There are not any deadlines on the response on these systems, but too long response times are annoying to the user. *Soft real-time systems* have deadlines but slightly missing one now and again does not cause any problems. Multimedia applications are examples of soft real-time. In *hard real-time systems* a deadline should never be missed. Missing a deadline could cause disaster. Medical equipment is an example of hard real-time systems.

The term “real-time garbage collection” is often used even for interactive applications. This is not the target of our research. The technique presented in this paper targets the hardest of real-time systems.

2.1 Scheduling Garbage Collection

Henriksson proposes a scheduling analysis that can be implemented using many different garbage collectors. An implementation using an incremental copying garbage collector based on Brook's algorithm [4] is presented in Henriksson's thesis [10]. The processes are divided into high and low priority processes. The high priority processes do a minimal amount of garbage collection work to increase their speed. The work is instead performed when entering the low priority processes and during their execution. To guarantee that high priority processes do not run out of memory, enough memory must be preallocated. Henriksson provides the analysis needed to calculate the amount of memory needed and the execution time of the garbage collector.

2.2 Real-Time GC in the Jamaica JVM

Siebert [13, 14, 15] proposes a combination of scheduling and splitting objects into equally sized blocks in an incremental mark sweep collector. The blocks are garbage collected as separate entities. Arrays are stored either continuous if continuous memory can be found fast enough, otherwise arrays are stored as a tree.

Siebert also provides an analysis to guarantee non-disruptiveness of the application. The input to the analysis is upper bound of reachable memory and the output is the number of GC increments needed per allocated block.

3 Eliminating External Fragmentation

In most systems fragmentation causes no problem [11], but in hard real-time systems it must be guaranteed that fragmentation does not cause the system to run out of memory. Fragmentation can be eliminated by compacting memory, e.g. using copying [9] or mark-compact [12] garbage collection, but that is often considered too expensive and/or unpredictable. Compaction also makes interfacing to other systems harder. Other solutions include prohibiting heap allocation or limit it to allocation into arenas that are deallocated as a whole.

The technique proposed in this paper is based on splitting objects into equally sized blocks as in many file systems and virtual memory systems. This eliminates external fragmentation, but introduces internal fragmentation of half a block size per object in average. An important difference between internal and external fragmentation is that internal fragmentation is predictable, but external is not. There are several ways to connect blocks into an object, e.g. linked list [15] and index blocks.

Below we assume that the blocks are kept in a linked list, but other techniques work as well. The requirement is that member access should be predictable, taking a step in the iteration through the blocks of an object should take constant time, and it must be possible do disconnect a block from the object in constant time.

3.1 Selecting Block Size

Selecting the size of the blocks is crucial to limit the loss in both execution time and internal fragmentation. You would like to find a size that makes most objects to fit in one or two blocks, but still does not cause the internal fragmentation to be too large.

The best block size depends on the application. The allocation function can be used to produce statistics about how many objects of different sizes are allocated. This information can be used to select block size. One should also consider the behavior of the cache when selecting the block size.

4 Worst-Case Execution Time

The increment operation of reference counting just increments a variable, and that is predictable. However, the decrement operation is potentially recursive. In the worst case a decrement operation can deallocate all objects on the heap, and even if that is bounded, it is too long for close to every systems. By using deferred reference counting [16] the decrement operation becomes constant in time. Using deferred reference counting, objects are added to a to-be-free list instead of decrementing its child references and deallocating the object (see Figure 2). The child references still need to be decremented, which now is done by the allocator instead.

```
algorithm decrrc(obj)
    if obj $\neq$ null then
        obj.rc $\leftarrow$ obj.rc - 1
        if obj.rc $=$ 0 then
            freelist.add(obj);
        end if
    end if
end
```

Fig. 2. Deferred reference counting

When the allocator needs more memory, objects are taken from the to-be-free list. Before they are reused their child references are decremented. Deferred reference counting assumes that all objects are of equal size, but that is not case in most systems. All blocks are of equal size, but the objects can contain multiple blocks. It is not desirable to decrement all child references of the next object in the to-be-free list if only a few blocks are needed for the new object. Thus, it must be possible to get a single block from the free list and this must be done in constant time.

We now have a system without external fragmentation with predictable allocation and reference count update operations. It is also guaranteed that all dead blocks are immediately available to new objects. If it is desirable to increase the speed of allocation in high-priority processes [10] this can be achieved by forcing the free list (which contains blocks which child references have already been decremented and initialized) to contain the number of blocks needed by the high priority process.

Two disadvantages remains in this proposal: the inability to reclaim cyclic data structures and the execution time overhead. Cyclic data structures can by reclaimed by using a backup garbage collector. The collector should be a non-moving incremental garbage collector, e.g. mark-sweep or the treadmill [1]. A disadvantage of using these techniques as backup is that both the reference counter and the garbage collector must be maintained. Thus, the execution time and memory overhead increases. Then why not skip reference counting completely? This is not a good idea, since no automatic memory manager except for reference counting can guarantee that dead memory is immediately available to the allocator. This is one of the main advantages to the technique presented in this paper. If this is not required, the treadmill can be used in combination with splitting objects into blocks. A disadvantage of using the treadmill as backup is that it requires two pointers in each object, which is quite expensive.

Splitting objects into blocks introduces an overhead in both memory usage and execution time. Using a linked schema each objects need a pointer to the next block. These pointers need to be traversed when accessing data in blocks other than the first. The number of pointers to traverse to get a field is known at compile time. Large objects such as arrays should not use a linked schema, because that would make the execution time of an indexing operation linear to its argument. Arrays can be stored as trees that make the execution time logarithmic to the size of the array. To reduce the execution time of reference counting, all reference counting optimizations can be used.

4.1 Comparison

As stated above, reference counting has some disadvantages to other garbage collection techniques. There do exist real-time copying and mark-sweep collectors, so the need for a real-time reference counting technique might be questioned. Table 1 compares the worst-case execution time of the operations of the real-time reference counter presented in this paper, to the worst-case execution times of other real-time garbage collectors. Neither the copying nor the mark-sweep algorithm use either increment or decrement operations. However, both use read/write barriers that perform equivalent operations. In the table the worst-case execution time of the barriers are compared to the worst-case execution time of the increment/decrement operations. The size of an object is denoted by s and the minimum amount of free memory denoted by f.

In real-time systems, the amount of memory is often limited. The real-time reference counting technique enables the usage of all of the memory. This is not possible using the other techniques due to the behavior of the allocation/free operations. This is especially advantageous in embedded systems.

	RT-Copying	**RT-Mark-Sweep**	**RTRC**
Increment (or equivalent)	$O(1)$	$O(1)$	$O(1)$
Decrement (or equivalent)	$O(1)$	$O(1)$	$O(1)$
Allocation/Free	$O(s+\frac{1}{f})$	$O(s+\frac{1}{f})$	$O(s)$
Member access	$O(1)$	$O(s)$	$O(s)$
Array access	$O(1)$	$O(\log s)$	$O(\log s)$

Table 1. Worst case execution time of operations in real-time garbage collectors

5 Implementation

A prototype of hard real-time reference counting has been implemented using the CPP macro preprocessor and a run time system implemented in C. The object splitting must currently be done manually.

In Figure 3 the types needed by the reference counter is defined. The type `type_t` declares a type containing type information. The only information needed be the reference counter is the size of objects of a type and a function which decrements a specific block of an object of the type. Next the type of a block is defined. It is defined as a union of the head of an object and a char array of `BLOCK_SIZE` cells. Here the block size is 32 but that is just a compile time constant. The head of an object contains a pointer to the next block in the object. This field also connects the blocks in the free list. Next is a union containing the reference counter or the next field used in the to-be-free-list. The reference count is always zero when the object is in the to-be-free-list, so it is not needed. Finally the type of an object is stored. Only the next pointer is used in blocks after the first in an object. Thus, the overhead is 12 bytes in the head block of an object and 4 in the following. The sizes can of course differ on other platforms.

The global state of the implementation is kept in the variables shown in Figure 4. Users of the reference counter should not touch these variables. The `freelist` keeps all blocks that are ready to be used by the allocator. These blocks should no longer refer to objects. Most systems would gain speed if all non-header cells where initialized to zero. The `available` variable keeps the number of blocks in the free list. The `tbflist` keeps the to-be-free list. Here all dead objects are listed before they are either moved to the free list or directly allocated to an object. Before the objects leave this list their references to other objects need to be decremented. The `head` variable points to the next block to be allocated or `NULL`. If the pointer equals `NULL`, the next object in the to-be-free list should be examined. The `type` variable keeps the type of the object whose blocks are currently being reclaimed. Variable `blockseq` is the sequence number of `head` in the current object. Finally an array called `heap` keeps all blocks available to the system.

Initiating the reference counter is just a matter of connecting the blocks into one huge free list. There is no need to initiate anything to zero since that is done automatically with global data. The function is shown in Figure 5 and its execution time is linear to the number of blocks on the heap.

```
#define BLOCK_SIZE 32

typedef struct type_t {
  size_t size;
  void (*decchildren)(struct type_t *t,
                      struct objhead_t *b,
                      int n);
} type_t;

typedef struct objhead_t {
  struct objhead_t *next;
  union {
    struct objhead_t *next;
    unsigned int rc;
  } nr;
  type_t *type;
} objhead_t;

typedef union block_t {
  objhead_t head;
  char data[BLOCK_SIZE];
} block_t;
```

Fig. 3. Type declarations

```
static objhead_t *freelist  = NULL;
static size_t     available = 0;
static objhead_t *tbflist   = NULL;
static objhead_t *head      = NULL;
static type_t    *type      = NULL;
static int        blockseq  = 0;
static block_t    heap[NUM_BLOCKS];
```

Fig. 4. Global data

```
void rc_init() {
  int i;

  for (i = 1; i < NUM_BLOCKS; i++) {
    heap[i-1].head.next
         = (objhead_t *) &heap[i];
  }
  heap[NUM_BLOCKS-1].head.next = NULL;
  freelist = (objhead_t *) heap;
  available = NUM_BLOCKS;
}
```

Fig. 5. Initialization

Allocating blocks from the free list is done by: traversing the free list until the requested number of blocks has been found, terminating the list of blocks, and adjusting `available` and `freelist`. This is shown in Figure 6. The execution time of the function is linear to the number of blocks being allocated.

```
static objhead_t *
alloc_from_freelist(int nb,
                    objhead_t **plast)
{
  objhead_t *bs = freelist;
  objhead_t *last;
  int n;

  available -= nb;

  for (n = 1; n < nb; n++) {
    freelist = freelist->next;
  }

  last = freelist;
  freelist = freelist->next;
  last->next = NULL;

  *plast = last;

  return bs;
}
```

Fig. 6. Allocating from the free list

Allocating from the to-be-free list is slightly more complicated. One object is taken from the list. The type information is stored in `type`, `blockseq` is set to zero, and `head` points to the first block of the object. For each block being allocated its child references are decremented using the `childdec` function stored in the type information. If we run out of blocks in the current object, the next object in the to-be-free list is taken and its blocks are used. The function is shown in Figure 7. The execution time of function is linear to the number of blocks being allocated.

The user function used to allocate objects allocates as many blocks from the free list as possible. The rest of the blocks are allocated from the to-be-free-list. Finally the lists of blocks are concatenated and the object is initiated. The function is shown in Figure 8 and its execution time is linear to the size of the object being allocated.

To guarantee a number of blocks in the free list the `rc_prealloc` function can be used. It simply allocates the necessary number of objects from the to-be-free-list and puts them into the free list. The function is shown in Figure 9, and

```
static objhead_t *
alloc_from_TBF(int nb, objhead_t **plast) {
  objhead_t *bs = head, *last = NULL;
  int n;

  for (n = 0; n < nb; n++, blockseq++) {
    if (head == NULL) {
      head = tbflist;
      if (head == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(5);
      }

      tbflist = tbflist->nr.next;
      type = head->type;
      blockseq = 0;

      if (last == NULL) {
        bs = head;
      } else {
        last->next = head;
      }
    }

    type->decchildren(type, head, blockseq);
    last = head;
    head = head->next;
  }
  last->next = NULL;
  *plast = last;

  return bs;
}
```

Fig. 7. Allocating from the to-be-free list

```
object_t *rc_alloc(type_t *t) {
  int nb;
  int fromfree;
  objhead_t *blocks, *tbfblocks = NULL;
  objhead_t *last, *dummy;

  nb = t->size;
  fromfree = MIN(nb, available);

  if (nb - fromfree > 0) {
    tbfblocks = alloc_from_TBF(nb - fromfree,
                               &dummy);
  }

  if (fromfree > 0) {
    blocks = alloc_from_freelist(fromfree, &last);
    last->next = tbfblocks;
  } else {
    blocks = tbfblocks;
  }

  blocks->type = t;

  return blocks;
}
```

Fig. 8. Allocating objects

its execution time is linear to the number of blocks requested. This function is typically called when leaving a high-priority process and start executing a low-priority process. The allocation function must keep the number of pre-allocated blocks in the free-list, so that they are available the next time a high-priority process starts executing. The only change needed in `rc_alloc` is to subtract the number of pre-allocated blocks from the number of available blocks when calculating `fromfree`.

```
void rc_prealloc(int nb) {
  objhead_t *bs, *last;

  if (available >= nb) return;

  bs = alloc_from_TBF(nb - available, &last);

  available = nb;
  last->next = freelist;
  freelist = bs;
}
```

Fig. 9. Pre-allocating blocks

Finally the function used to put objects onto the to-be-free list called by the decrement operation is presented. This function should be inlined. On a Pentium processor it expands to three assembler instructions. The function is shown in Figure 10 and its execution time is constant.

```
void rc_release(objhead_t *b) {
  b->nr.next = tbflist;
  tbflist = b;
}
```

Fig. 10. Releasing objects

The decrement and increment operations are implemented according to the algorithms presented in Figure 1 and 2. The execution times of both operations are constant.

5.1 Summary

The overhead per object (on x86-Linux-2.2 using gcc-2.96) is 4 bytes for reference counter/next pointer in free lists and 4 bytes for type information plus 4 bytes per used block. For most objects this should give a total overhead of 12 – 16 bytes.

Initiating the memory manager is a matter of putting all blocks in a linked list. The worst-case execution time is linear in respect to the size of the heap.

Allocation (from the to-be-free-list or free-list) takes blocks from respective list. When blocks are taken from the to-be-free-list their references must also be released. Since releasing references of a block takes constant time (the size of blocks are constant and thus the maximum number of references), the worst-case execution time of an allocation is linear in respect to the size of the allocated object.

In some systems it is desired that allocation in high-priority processes should be as fast as possible. This can be accomplished by pre-releasing references of blocks in the to-be-free-list and moving these to the free-list. When a high priority process starts executing it must be guaranteed that the number of blocks needed by the high-priority process is in the free-list. This is done by the `rc_prealloc` function. The worst-case execution time of this function is linear in respect to the number of blocks that should be pre-released.

Finally, both the increment and decrement operation have a constant worst-case execution time.

6 Benchmarks

To conduct an evaluation of the technique, a small simulation application has been implemented. The application simulates two water tanks. A pump pumps water in to the upper tank, which in turn is connected to the lower tank via a pipe. The lower tank has a hole in the bottom. The application aims to keep a specified level in the lower tank by controlling the pump. The simulation consists of five objects: two tanks and three flows. The system has been tested in 12 variations with and without: splitting objects in two blocks (of 64 bytes each), separating the blocks of an object with half the heap size, reference counting, and running thousand simultaneous simulations. The test was performed by running the system for 40 000 000 iterations, except when thousand simulations where run simultaneously then 40 000 iterations where run. Every variation where run five times on a bare Linux machine. The shortest execution times of each variation are presented in Figure 11.

The conclusion from these tests is that splitting objects is expensive when using more memory. Choosing a different block size might improve this behavior. It is remarkable that reference counting is almost for free in the tests. This is most likely due to that the floating-point calculations in the simulation hide the effect of the reference count updates. Over 24 million increment and decrement operations where performed in every run of the system (with reference counting enabled).

7 Conclusion and Future Work

This paper presents an automatic memory management technique for hard real-time systems. It does not suffer from external fragmentation, all opera-

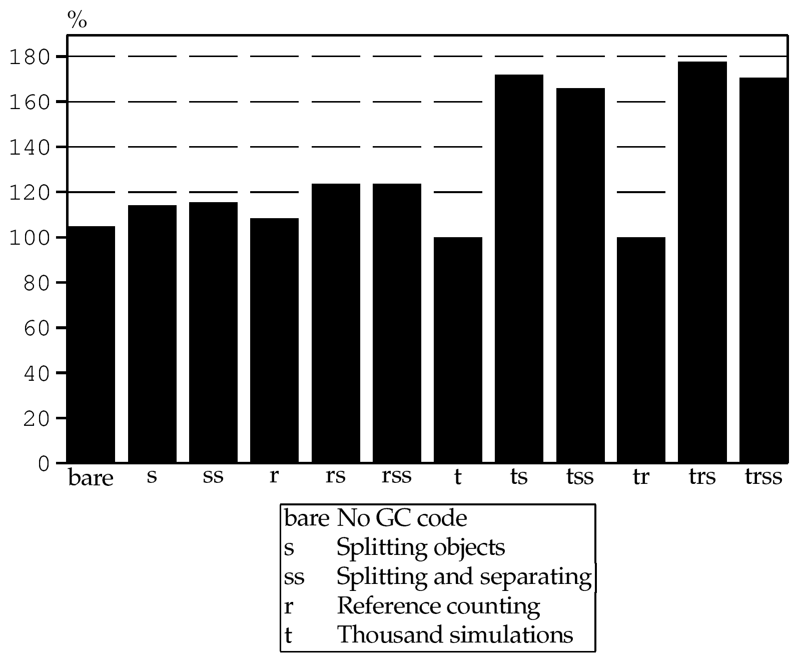


Fig. 11. Execution time in percent compared to a bare run

tions are predictable in memory usage and execution time. All (non-cyclic) dead blocks are immediately available to new objects. And finally, the high priority processes can be given the benefit of faster allocation if that is desired. Thus, the presented technique is fully predictable.

The remaining disadvantages are the inability to reclaim cyclic data structures and the execution time overhead.

Currently we have to use a backup garbage collector to reclaim these. A backup garbage collector can be run continuously or it can be activated at specific times. What to choose depends on how much cyclic garbage is produced. This might not be a good solution if much cyclic data is produced, but in most systems cyclic garbage can be avoided. One could use weak pointers that are not counted, or one could break cycles manually before they become garbage. A good idea would be to run a backup garbage collector during development.

The execution time overhead can be greatly reduced using static optimizations. The optimizations can turn heap allocations into stack allocations and remove redundant reference count updates.

Next this technique will be implemented in a Java^{TM}-to-assembler compiler. This will provide the figures for execution time and memory usage overhead, which will be used to tune the implementation.

Looking on how to connect blocks into objects, choose block size, integrate a backup garbage collector, and what to do with large objects will then continue this work. Another topic is to optimize the reference counter. This is done in parallel with this work.

References

[1] Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.

[2] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[3] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[4] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[5] David R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, Nancy, France, September 1985. Springer-Verlag.

[6] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[7] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[8] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[9] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[10] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[11] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[12] Robert A. Saunders. The LISP system for the Q–32 computer. In E. C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 220–231, Cambridge, MA, 1974. Information International, Inc.

[13] Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 130–137, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[14] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[15] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Proceedings of Compilers, Architectures and Synthesis for Embedded Systems (CASES'00)*, San Jose, November 2000.

[16] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

Session II

Static Program Analysis And Optimizations

Flow Sensitive Call Graph Construction For Java

Christian Probst*

FR Informatik
Universität des Saarlandes
probst@cs.uni-sb.de

Abstract. The Joses project is going to develop techniques and tools to enable the use of Java for embedded systems. The necessary tasks include developing analyzers for object oriented languages. One problem in analyzing these languages is that the exact control flow graph is not known statically due to dynamic method dispatching. This, however, is needed in order to apply the large class of known inter-procedural analyses. Control flow analysis generally aims to determine which closures appear at which points in the execution of the program. For object oriented languages this means to calculate possible target classes for method calls. In contrast to the widely spread techniques, we will present a flow-sensitive approach.

1 Introduction

Object oriented languages are more difficult to translate into efficient programs than traditional imperative languages such as C. A major reason for this are the different levels of optimizations applied to these classes of languages. One key problem is the lack of an exact control flow graph at compile time.

However, this graph is required for most traditional data flow analyses, which rely on the ability to identify all possible successors or predecessors at every program point. With procedures as parameters and method calls depending on the actual object stored in a variable (dynamic dispatch) this identification is no longer possible.

Control flow analysis solves the problem by computing for every subexpression a number of functions or program points it may evaluate to at runtime. Having this information one can for every function application determine its successors, thus enabling the projection of traditional techniques onto higher order languages. While control flow analysis is usually believed to be rather runtime consuming, recent experiments suggest that this might not be completely true. Rather it looks like the additional runtime seems to be acceptable taking into account the analysis' precision.

Nevertheless, only flow- and context-insensitive approaches are widely spread today. In this paper we present a flow-sensitive approach that may easily be

* Supported by the Esprit LTR project #28198 "JAVA and CoSy Technology for Embedded Systems"

extended to include context-sensitivity. The standard approach is to add conditional constraints at method call points. However, this may mean that a lot of constraints are added that might never be evaluated. The concept we present will allow for constraint addition on demand only for those methods that really will be called.

In what follows we will first give a brief sketch of control flow analysis. After that we will show a simple example, give the rules how to generate the constraints, and show how to solve them.

`rtype m() { ··· }`	*method definition*
`rtype m(p) { ··· }`	*method definition*
`v1 = v2`	*assignment*
`v = new C()`	*object creation*
`v = o.m(a)`	*method call*
`return v`	*return*

Fig. 1. Statements in the example program

2 Control Flow Analysis

Classical control flow analysis (CFA) computes for each subexpression the functions that it may evaluate to. Thereby it is possible to determine where the flow of control may be transferred to at that point. This means that control flow analysis in its usual area of analyzing functional languages is *functional* in the sense that it takes functions as abstract values and traces them through the program being analyzed. The analysis itself is similar to the computation of *definition–use–chains (DU)* known for imperative languages in the sense that both trace how definition points reach points of use [3]. For object oriented languages CFA traces sets of either classes or objects a variable may point to at runtime. The point of usage, however, is still function application or, in object oriented words, method send.

Different kinds of CFA may be distinguished based on the amount k of context they take into account [3, 2]. This can be compared with the call-string approach used in traditional data flow analysis and allows the analysis to remember the last up to k dynamic call points.

In order to compute the CFA-solution we need to label method parameters, variable occurences, and ends of procedures with unique tags $l \in \mathbf{Lab}$ (the superscripts in figure 3). The example is stated in a subset of Java using the statements given in figure 1. Labels are assigned to all occurences of variables, parameters, method ends and expressions.

In figure 2 we define a 0–CFA. The abstract values it propagates through the program are elements of $\mathbf{Val} = \mathcal{P}(Classes)$, i.e. sets of classes. $\mathbf{Cache} = \mathbf{Lab} \rightarrow \mathbf{Val}$ is the abstract cache, associating the program labels with abstract values. The result of the control flow analysis then is $C_\star \in \mathbf{Cache}$ that associates a set of classes with each labelled program point.

$$
\begin{array}{ll}
[\mathit{variable}] & C_\star[v^l] := \bigcup_{l' \in DU(v,s)} \{C(l') \subseteq C(l)\} \\
[\mathit{assignment}] & C_\star[v^{l_1} := \mathit{expr}^{l_2}] = C_\star[\mathit{expr}^{l_2}] \cup \{C(l_2) \subseteq C(l_1)\} \\
[\mathit{creation}] & C_\star[(\mathit{new}\ C())^l] := \{\{C\} \subseteq C(l)\} \\
[\mathit{return}] & C_\star[\mathit{return}\ v^l] := C_\star[v^l] \cup \{C(l) \subseteq C(l_m)\} \\
 & \qquad \text{where } l_m \text{ is the end label of the method body analyzed} \\
[\mathit{noarg}] & C_\star[(v^{l_1}.m())^{l_2}] := \mathit{CALL}_m(l_1, l_2) \\
[\mathit{arg}] & C_\star[(v^{l_1}.m(a^{l_2}))^{l_3}] := \mathit{CALL}_m(l_1, l_3, l_2) \\
[\mathit{method\_noarg}] & \mathit{rtype}\ m()\ \mathit{body}^l : \mathit{ConstraintSet}(\mathit{id}(m)) := (l, []) \\
[\mathit{method\_arg}] & \mathit{rtype}\ m(p^{l_1})\ \mathit{body}^{l_2} : \mathit{ConstraintSet}(\mathit{id}(m)) := (l_2, [l_1])
\end{array}
$$

Fig. 2. Constrained based 0–CFA

Let's have a look at some of the rules in figure 2. The [*variable*] clause means that an expression consisting of a variable may evaluate to all the abstract values associated with definitions reaching this program point. For the assignment of an arbitrary expression to a variable ([*assignment*]) we first evaluate that expression ($C(l_2)$). This set must be assignable to the variable x, so $C(l_2) \subseteq C(l_1)$ must hold.

One of the more interesting clauses is [*arg*], i. e. a method call with an argument. The object (x^{l_1}) and the argument (a^{l_2}) must be analyzable. Performing these analyses results in $C(l_1)$ and $C(l_2)$ holding the respective values. These will be accessed by the $CALL$ construct which will be introduced in section 3.2.

2.1 Preprocessing

Before generating constraints we will need to perform one further step – for each variable v and each program point p we compute the labels of definitions of v that are visible in p. This is a simple intraprocedural definition-use analysis, computing $DU(v,p) := \{l \in LAB \mid \text{label } l \text{ is a definition of } v \text{ visible at } p\}$. Thus for each variable v the computed value is either a set of labels contributing to the set of v or it is identified as related to a method call (i.e. it is the result of a call, a parameter or the end of a method). For our running example in statement s the variables va and vb are used. The computed DU-sets are $DU(va,s) = \{16\}$ and $DU(vb,s) = \{20\}$. After this we know for each labeled use of a variable which labels may contribute to its set.

3 Generating Constraints

3.1 Generating the intraprocedural constraints

Generation of interprocedural constraints is fairly easy - just add the constraints according to figure 2. In order to be prepared for handling calls we will store information related to methods by defining a function $\mathit{ConstraintSet} : \mathit{Methods} \rightarrow \mathbf{Lab} \times \mathit{list}(\mathbf{Lab})$. The first element is the label of the method's end point, the second is the list of labels of the method's parameters.

```
class A {
  A a;
  void set (Object this^1,A x^2 ) {}
  void print (Object this^3) {}
}
class B extends A {
  void print (Object this^4) {}
}
class C extends A {
  void print (Object this^5) {}
  void set (Object this^6,A x^7 ) {}
}
```

```
class test {
  A generate (test this^8) {
    A t^9=(new B())^10;
    return t^11;
  }^12
  void main (test this^13) {
    A va^14 = (new A())^15;
    B vb;
    va^16 = (this^17.generate(this^18))^19;
    vb^20 = (new C())^21;
    (va^22.set(va^23, vb^24))^25;         //s
    (va^26.print(va^27))^28;
  }^29
}
```

Fig. 3. Example program

3.2 Adding procedure calls

In figure 1 we have already given the rules how to label occurences of parameters and end of procedures. Of course this is not enough, as we will have to let flow

- the values of actual arguments of a procedure to its parameters
- the values associated with the procedure's endlabel to the label of the call expression.

$id(generate) = 5$, $id(main) = 6$

$ConstraintSet(5) = (12, [8])$	$\{B\} \subseteq C(10)$	$C(10) \subseteq C(9)$
	$C(9) \subseteq C(11)$	$C(11) \subseteq C(12)$
$ConstraintSet(6) = (29, [13])$	$\{test\} \subseteq C(13)$	$C(13) \subseteq C(17)$
	$\{A\} \subseteq C(15)$	$C(15) \subseteq C(14)$
	$C(13) \subseteq C(18)$	$CALL_{generate}(17, 19, [18])$
	$C(19) \subseteq C(16)$	$\{C\} \subseteq C(21)$
	$C(21) \subseteq C(20)$	$C(16) \subseteq C(22)$
	$C(16) \subseteq C(23)$	$C(20) \subseteq C(24)$
	$C(16) \subseteq C(26)$	$CALL_{set}(22, 25, [23, 24])$
	$C(16) \subseteq C(27)$	$CALL_{print}(26, 28, [27])$

Fig. 4. Constraints and *ConstraintSet* for the example program

The standard approach is to add conditional constraints, one for each method that might be callable at run time. However, this may mean that a lot of constraints are added that will be evaluated but may never contribute to their target sets. The concept we present will allow for constraint addition on demand only for those methods that really will be called. As already shown in figure 2 we will arrange for this by adding a macro constraint $CALL_m$ that will serve as a template for the constraints associated with procedure m. $CALL_m$ gets as its arguments

- the label l_v of the variable that contains the object whose m is called
- the label l_r to which m's result is going to be stored
- the labels l_i of all arguments (if any) that are passed to m

That is, in the example program the constraints for method *main* would be extended by $CALL_{set}(11, 13, 12)$ and $CALL_{print}(14, 15)$. When evaluating the macro during constraint solving care will have to be taken that only callable methods are selected, that constraints are added for letting the arguments flow into the parameters, and for letting the procedures result flow back to the label of the method calling expression.

3.3 Adding global variables

For the analysis of global variables things are slightly more complicated. For a use of such a variable the associated definition might be located in another procedure. Thus we will need to extend the information in the $CALL$ macro with information about the definitions of global variables currently visible. This can be done in a way similar to that how we computed the definition information - we simply run a use-definition analysis using the already computed DU-sets. We now get for every global variable gv and every program point p a set $UD(p, gv)$ that gives for each method entry point the first uses of gv. Thus when evaluating the $CALL_m$ macro for a call in statement p we will be able to connect the reaching definitions of gv as known from the $DU(p, gv)$ with the first uses of gv in the method m.

4 Solving the constraints

Having created the constraints for all the methods in a program, we will now start to solve them. In order to do so we first transform them into a graph and then let the solver work on this representation. For the example program the construction of $C_\star$ results in the function and the constraints given in figure 4.

4.1 Graph generation

Having computed $C_\star$ we construct a directed graph following [3]. That is for each $C(l)$ and $CALL(\cdots)$ a node n_l respectively $n_{CALL()}$ is created. For each constraint $C(l_1) \subseteq C(l_2)$ an edge from n_{l_1} to n_{l_2} and for each $CALL(obj, res, [args])$ an edge from each of $\{n_{obj}, n_{args}\}$ to $n_{CALL(\cdots)}$ is created.

With each node n a field $D[n_l]$ is associated, which will hold the computed set of classes for n_l. It is initialized by the set $\{t | \{t\} \subseteq C(l) \in C_\star\}$. For the example program the generated graph is given in figure 5. We have left out the annotation of edges with the associated constraint as this is evident from the nodes being connected.

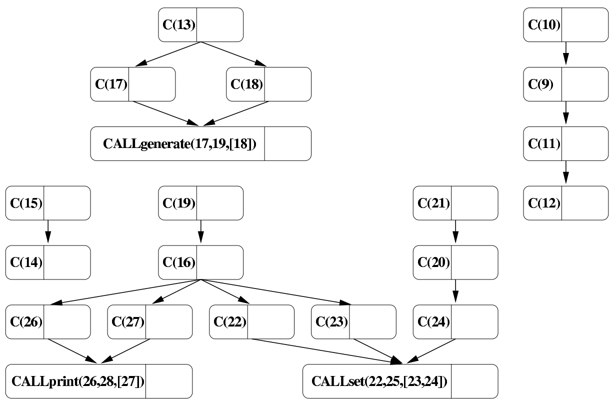


Fig. 5. The start graph for the solver

4.2 The solver

The solver works on a worklist that is initialized from the constraints for the program's entry points with those nodes n for which $D[n] \neq \emptyset$. Due to space limitations we will only sketch an outline of the algorithm, it is implemented along ideas of [3, 6, 4].

The solver iterates over the nodes in the worklist until the list is empty. Depending on the type of the selected node n and its successors, several actions may be taken.

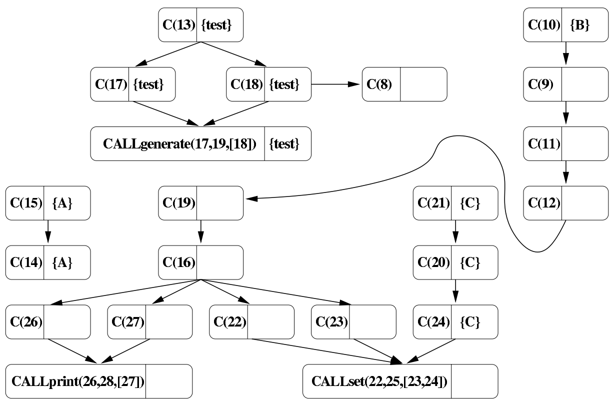


Fig. 6. Constraint graph before and after (dashed edges) visiting the first CALL node

If n is a node $C(l_1)$ and the successor is a node $C(l_2)$ then the constraint on the edge (n_{l_1}, n_{l_2}) is evaluated and $D[l_2] := D[l_2] \cup D[l_1]$. If $D[l_1] \cap D[l_2] \neq \emptyset$ then n_{l_2} is added to the worklist. Figure 6 represents the situation after all *normal* propagations have taken place. Note that the graph on the right side has not been proceeded as it represents the constraints of method *generate* which has not yet been called.

If n is a $CALL_m(obj, res, [args_j])$ node it represents a method call on the object labelled with obj. The solver then looks up the current set of possible classes, instances of which may be stored in the variable, and computes the identifiers of such methods that may be called. This computation is performed by inspecting the class hierarchy graph [5]. Using this information it looks up in the function $ConstraintSet$ the stored values for the callable implementations of m. Thus it obtains a set of pairs $t_i = (endlabel_i, [param_i])$.

For the call $CALL(13, 15, [14])$ the object to which the message is sent has type $\{test\}$, thus the call goes to the method *generate* in class *test*. Thus $t_i = (10, [6])$.

First an edge $n_{endlabel_i}$ to n_{res} is added, as the information computed for the end point of the procedure will have to be stored in the label of the method call expression. Furthermore for each argument and parameter pair $(args_j, param_i)$ an edge $(n_{args_j}, n_{param_i})$ must be added. In addition, one must assure that all uses of global variables in the added procedure are connected to the current visible definitions.

The new constraint graph is given in figure 6 by the dashed edges. The new edges resemble that the value of the call's argument (label 18) flows to the method's parameter (label 8) and that the method's result (label 12) flows to the method call expression (label 19).

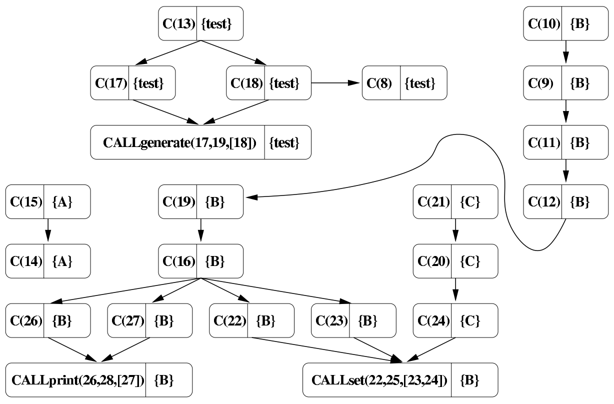


Fig. 7. The final constraint graph

Having added those edges worklist iteration continues, ignoring the calls to *print* and *set* assuming that they may not contain access to object variable. It finally results in the graph given in figure 7.

5 Evaluation

The $D[.]$ fields of the $CALL$ nodes tell us, which classes the object at the specific call site may have at run time. As one can see in figure 7, each object may only have one type, namely *test* in the call to *generate*, B in the call to *print* and B

in the call to set. By inspecting the class hierarchy one finds out that each of the calls has got an unique target - the impletation of *generate* in *test*, *set* in A and *print* in B. The following table gives the computed number of classes for the object at the call site. It compares our numbers (CFA_0 with results from other techniques, namely class hierarchy analysis (CHA, [5]), raid type analysis (RTA, [1]) and variable and declared type analysis (VTA, DTA, [7]). The differences mainly result from the other techniques being not flow sensitive.

call to	CHA	RTA	VTA	(DTA)	CFA_0
generate	1	1	1	1	1
set	2	2	1	1	1
print	3	3	2	2	1

6 Conclusions

We have presented a flow-sensitive approach to control flow analysis. By introducing a macro concept for method call handling we get rid of a potentially great set of conditional constraints. The macros allow for a constraint generation that occurs on demand only for those methods that may really be called. What needs to be done in the future is to add full support of global variables.

References

1. D. F. Bacon. *Fast and Effective Optimzation of Statically Typed Object–Oriented Languages*. PhD thesis, University of California, 1997.
2. The Joses Consortium. Control flow analysis – an overview. *D3.2.1, JOSES-8602-cfaintrocp*, 1999.
3. H. Nielson F. Nielson and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
4. C. Fecht and H. Seidl. An even faster solver for general systems of equations. Technical report, Universitaet Trier, 1996.
5. D. Grove J. Dean and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *LNCS 952*, 952, 1995. ECOOP'95.
6. H. Seidl and C. Fecht. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. Technical report, Universitaet Trier, 1997.
7. Laurie Hendren et al. Vijay Sundaresan. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, October 2000.

Static Method Call in Java

Holger Dewes, Christian Probst
email: {dewes,probst}@cs.uni-sb.de

FR Informatik
Universität des Saarlandes

Abstract. Dynamic method call is one of the major advantages gained by object-oriented languages. However, it entails severe performance penalties. Static program analysis is used to identify cases where dynamic dispatch can be replaced by static binding. This is done by computing the dynamic type of method invocation expressions. Whenever such an expression may only call one method, the dynamic call can be replaced by a static call. Unlike most approaches, we present a data flow analysis allowing the flow- and context-sensitive approximation of types of expressions. We start with a conservative approximation of the control flow graph constructed by means of standard techniques.

1 Introduction

Object-oriented programming languages like C++ and Java have become very popular in the last few years. Unfortunately, in general object-oriented programs are slower and bigger than programs written in pure imperative languages like C or Fortran. This is especially true for Java which offers many features that make Java programs simple and secure, but also slow.

One of these features is dynamic method call, a typical feature of object-oriented languages involving inheritance. A class may redefine any method inherited from its superclass. When calling a method, the *dynamic type* of the method invocation expression decides which of the (possibly many) applicable method definitions will actually be called.

Dynamic method call is usually implemented using *virtual function tables* (vtbls). Every class has its own vtbl with a pointer to the appropriate definition for every method visible in that class. When a class redefines an inherited method, the pointer in the vtbl is changed to the new definition. At runtime, the vtbl is used to look up the correct method definition based on the dynamic type of the method invocation expression.

Method calls using virtual function tables are more expensive than direct function calls for a number of reasons. The pointer has to be looked up, optimizations like method in-lining are not possible. These problems are the reason why it is desirable to compute the dynamic type of method invocation expressions at compile time.

The rest of the paper is organized as follows. In section 2 we give a short introduction to data flow analysis and the program analyzer generator PAG. In section 3 we give a description of the problem the analysis wants to solve. In section 4 we present the analysis in some detail.

2 Data Flow Analyses with PAG

The analysis presented in this paper uses the PAG [3] program analyzer generator to define a data flow analysis computing the dynamic type of method invocation expressions. Data flow analyses are a standard tool for compiler optimizations [4]. They are based on the *control flow graph* (CFG) representation of a program. The CFG consists of nodes for atomic statements and edges representing the possible flow of control between statements. The analysis associates each edge with an element of a domain, representing the information computed by the analysis. A transfer function is defined for each node, specifying how the information is changed by the statement represented by the node.

PAG is a tool for specifying and generating program analyzers. It offers an ML-like specification language for specifying the domain, the transfer functions, and all other information necessary to define an analyzer.

One special feature of PAG is the ability to generate *interprocedural* analyzers. Interprocedural analyses can cope with data flow between functions, whereas *intraprocedural* analyses can only analyze functions separately.

To allow data flow between functions, *call edges* are created from a call site to the function. To bypass caller information that is unchanged or should be invisible in the called function, *local edges* to special PAG return nodes are used. The data flow at function return is implemented by *return edges.*

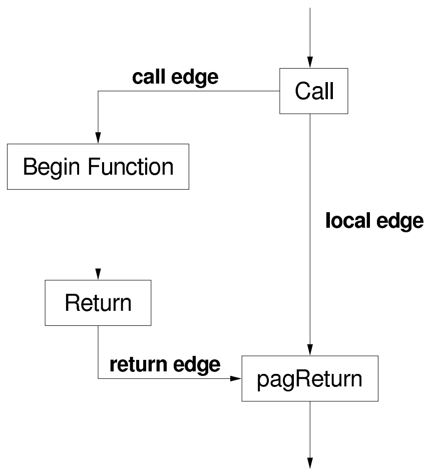


Fig. 1. Interprocedural control flow graph.

Figure 1 shows part of a control flow graph with a call node, a PAG return node, a function, the call edge, the local edge, and the return edge.

3 The Problem

As mentioned in the introduction, the goal of the analysis is to approximate the *dynamic type* of method invocation expressions. The dynamic type is defined as the class of the reference the expression evaluates to during runtime. This information can then be used to limit the set of applicable method definitions as determined by the static type of method invocation expressions at call sites. Ideally, only one method definition can be called and the dynamic method call can be replaced by a more efficient direct function call.

The elimination of an applicable method definition corresponds to the elimination of the call edge to that particular method definition in the CFG. Thus, the information computed by the analysis can also be used to create a more precise CFG to improve subsequent data flow analyses.

The problem can be stated more precisely as follows:

> Given a method invocation $e.m(\ldots)$ at a specific program point, what is the set of types of the references the expression e can evaluate to at runtime?

Once this set is computed, the set of applicable method definitions can be easily determined.

To analyze a method invocation expression, one must first ask how such an expression can look like. The Java Language Specification [2] gives the following answer.

MethodInvocation:
  MethodName (ArgumentList$_{opt}$ *)*
  Primary . Identifier (ArgumentList$_{opt}$ *)*
  super . *Identifier (ArgumentList*$_{opt}$ *)*
  ClassName . super . *Identifier (ArgumentList*$_{opt}$ *)*

Method calls involving the keyword *super* statically determine the appropriate method definition. They can be resolved at compile time and are thus not interesting for the analysis. The first case is for instance methods semantically equivalent to

  this . *MethodName (ArgumentList*$_{opt}$ *)*

which is covered by the second case.

The definition of *Primary* is too complex to give in full detail here. Basically it covers simple expressions, i. e. literals, class literals, the keyword *this*, field accesses, array accesses, and method invocations.

For the analysis this means that the following language constructs need to be analyzed: variables, class fields, instance fields, arrays, and method invocations.

When the set of dynamic types of the above constructs is known, the dynamic type of any arbitrary method invocation expression can be determined.

4 The Analysis

4.1 The Control Flow Graph of a Java Program

Before any data flow analysis can be started on a given program, the CFG for this program needs to be constructed. For a Java program this is rather straightforward, with the exception of dynamic method call sites. In those cases, the set of applicable methods as target of call edges needs to be conservatively approximated, i. e. any method definition that can be called at runtime must be an element of the set.

There are many efficient algorithms to construct a conservative CFG. We use rapid type analysis [5] which is simple and efficient and provides a good starting point for the analysis.

4.2 The Domain

The main aim of the analysis is to collect all references that may be assigned to locations, i. e. to variables, fields, and arrays. Every location is therefore mapped to a set of references.

The definition of the domain of the analysis is as follows.

$$(str \times bool) \rightarrow \mathcal{P}(unum \times snum \times unum \times str)$$

We use a functional domain. As stated above, each function maps IDs of locations to a set of abstractions of references. The location ID consists of a unique ID string (of string type *str*) and a flag (of boolean type *bool*) indicating whether the location is a local variable or a field. This is needed for correctly handling the interprocedural flow of data.

The reference abstraction consists of four parts. The first part is the ID of the node where the reference was assigned to the location (an unsigned number of type *unum*). This is used mostly for debugging the analyzer.

The second part of the reference abstraction is the context number, a signed number of type *snum*. This is one attribute used to distinguish references. The context number stems from the way PAG handles interprocedural data flow. A standard way of tracking the context of a function and to thus distinguish instances of function calls are *call strings*. A call string is a string of call nodes. The last part of the call string is the node that called the current function, the part before is the node that called that function and so on. The different call strings of a function are enumerated by PAG, and these numbers are the context numbers. The analysis uses these context numbers to distinguish between references created at the same program point but in different contexts of the function. The maximum length of the call string can be set by the user of the analysis. Longer call strings result in higher precision, but also in higher costs.

The third part is the ID of the node where the reference was created and is of type *unum*. This is the main attribute for distinguishing references: if two references are created at two different program points, they are treated as different; if two references are created at the same program point *and* in the same context, they are treated as being identical. The reason why we use abstractions of references instead of tracking every reference individually lies in the nature of data flow analyses. Iterations of loops e. g. are only repeated until a fixed point is reached. This way, the analysis will always terminate, but may not identify each iteration of the loop individually.

The fourth and final part is the dynamic type, i. e. the class name of the reference, of type *str*. This is the information needed to determine the dynamic type of method invocation expressions as explained before.

4.3 The Identification of Locations

For greatest precision it is important to track every location individually, if possible. For class fields this is easy. Class fields are statically accessed using class names. They are global locations that exist as long as the class they belong to exist. The identification string for class fields consists of the class name and the class field name. This way, any class field can be uniquely identified.

Variables, i. e. (method) local variables and parameters, can also be uniquely identified. They have a unique identification number in the internal representation (IR) that is also used by the analysis. However, variables are objects on the stack. As such the analysis does not track every instance on the stack of a specific variable individually. Instances of method calls are only distinguished as specified by the context numbers as explained above.

It turns out that instance fields cannot be identified individually. Instance fields are accessed the same way as instance methods: through expressions of reference type. To track every instance individually means identifying every reference individually. As explained above, this is not possible. Instead, the same abstractions of references as used for the dynamic method calls are used. The identification string for instance fields consists of the context number, the creation node ID, and the class name of the reference that was used to access the field when its value was assigned to it, plus the field name.

Arrays can be seen as a special kind of instance field. In Java, arrays are objects which support the operator `[]` for accessing the elements of the array. Array references are therefore handled the same way as other object references. Furthermore, the analysis does not distinguish the elements of an array but treats them all together as one. The identification string of arrays is the same as for instance fields except that it does not contain a field name.

Obviously the analysis loses some precision when dealing with certain kinds of locations. Some precision is lost when dealing with local variables: different call instances of the variable's method may not be distinguished. However, since the analysis eventually computes only one set of types for every program point, but not for every call instance, this only affects the precision of the data flow. More precision is lost when analyzing instance fields and arrays. Fields of instances created at the same node and in the same context will not be distinguished. Whether this is a problem depends on the nature of the program. Very often instances created at the same program point will also have similar properties and thus the imprecision should not matter. Finally, we believe that ignoring the index of arrays should not be a major problem. The elements of an array are usually accessed within a loop, which means that all elements are accessed at the same program point, for which, again, only one set of types is computed anyway.

4.4 The Analysis of Assignments

As an example for a transfer function we now present in some detail how assignment statements are handled by the analysis. Let the assignment be $le = re$.

First the right hand side expression re is analyzed. The support function `getClassList` is used to get the list of references (possibly) rendered by re. This is the value that is being assigned to the left hand side expression le. Three different cases are considered for le.

Variable or Class Field If le is a local variable, a parameter, or a class field the identification string is constructed and then updated to the new value given by re. The update is destructive, i. e. the old value is destroyed.

Instance Field If le is an instance field access of the form $e.f$, first the set of references given by e is computed by `getClassList`. Then for every element C in this set, the appropriate update for $C.f$ is executed. The update for instance fields may not be destructive. This is because of the impreciseness of the analysis when handling instance fields. At runtime, only one reference is rendered by the expression e, and only the field f of that reference is changed. However, the analysis may compute redundant references for e. The field f of these references, however, will not be changed, and therefore the information associated with these references may not be destroyed.

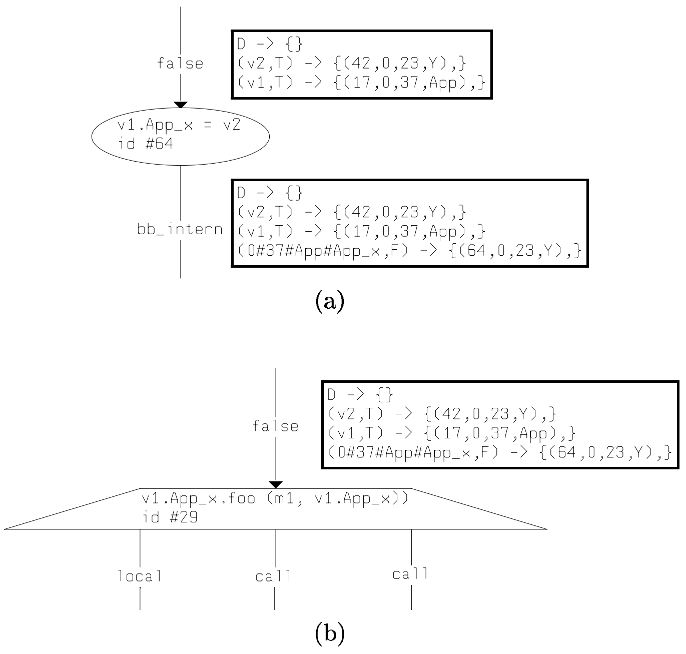
Array Arrays are basically handled the same way as instance fields. The only difference lies in the fact that an array has no fields and all elements of the array are treated the same.

Note that method invocations cannot be part of an assignment. The front-end extracts any method invocation from expressions and creates separate statements for them.

Example Consider the following Java program.

```
class X                            class App {
{ public void foo(){} }                private X x;
                                       public void m() {
class Y extends X                          x = new Y();
{ public void foo(){} }                    x.foo(); } }
```

In figure 2(a), the analysis result for the `x = new Y();` statement of method *App.m*() is given. For simplicity, we use names instead of numbers for variables and only show the relevant information. The variable *v1* holds the value of the *this* pointer, the variable *v2* holds the result of the `new Y()` expression. After



false
D -> {}
(v2,T) -> {(42,0,23,Y),}
(v1,T) -> {(17,0,37,App),}
v1.App_x = v2
id #64
bb_intern
D -> {}
(v2,T) -> {(42,0,23,Y),}
(v1,T) -> {(17,0,37,App),}
(0#37#App#App_x,F) -> {(64,0,23,Y),}
(a)
false
D -> {}
(v2,T) -> {(42,0,23,Y),}
(v1,T) -> {(17,0,37,App),}
(0#37#App#App_x,F) -> {(64,0,23,Y),}
v1.App_x.foo (m1, v1.App_x))
id #29
local
call
call
(b)

Fig. 2. (a) assignment `x = new Y();` (b) method invocation `x.foo()`

the assignment, the field *App_x* as accessed through the reference in *v1* holds the value of *v2*.

Figure 2(b) shows the result for the method invocation `x.foo()`. The method invocation expression *v1.App_x* is an instance field with the dynamic type *Y*. Therefore, the call edge to method *X.foo*() can be eliminated and the method invocation can be replaced by a direct function call to method *Y.foo*().

4.5 The Interprocedural Part of the Analysis

So far we have only covered the treatment of locations. However, the analysis of values returned by method calls are just as important. By using PAG, most of the work needed for interprocedural analysis is done by the generator. The designer of the analysis only needs to decide which data to send over each of the call edges, local edges, and return edges.

The data of a location is sent over the call edge if it is visible outside of the current method. That is true for fields only. Local variables and parameters are therefore sent over the local edge. As already mentioned, the boolean flag of every location is used to make that decision.

In the case of the return edge, the expression associated with the *return* statement is analyzed and the appropriate data is sent back over the return edge.

4.6 Restrictions

It is vital for the correctness of the analysis not to miss any reference creations and assignments. Therefore, the analysis needs to see the complete code of a program, i. e. it is a whole-program analysis.

The analysis cannot cope with multi-threaded programs.

5 Related Work

Several flow- and context-insensitive analyses are presented in [5], including rapid type analysis (RTA). These analyses use different strategies to improve precision, such as computing one set of types for every method instead of just one set for the whole program as RTA does. However, our analysis is still more precise than each of these as it computes one set of types for every expression.

Another analysis presented in [5], k-CFA, is an example for a context- and flow-sensitive analysis, which also computes one set for every expression. However, it is still more imprecise than our analysis because it abstracts objects to their classes, while our approach abstracts objects to their creation-node and -context. This allows for a more precise analysis of instance fields.

6 Conclusion

We have presented a flow- and context-sensitive analysis which computes the set of references stored in locations and returned by method invocations. The result of the analysis is used to approximate the dynamic types of method invocation expressions and replace calls to only one method definition with a static call.

7 Acknowledgements

The work presented in this paper was supported by the Esprit LTR project #28198 "JAVA and CoSy Technology for Embedded Systems." The analysis has been implemented in the CoSy compiler construction framework [1].

References

1. Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy compiler phase embedding with the CoSy compiler model. In *5th International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 1994.
2. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* The Java Series. Addison-Wesley, Reading, MA, 2000.
3. Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
4. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, Berlin;Heidelberg;New York, 1999.
5. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, October 2000.

Static Scheduling of Sequential Java Programs for Multi-processors

Peter Aronsson,Peter Fritzson
Dept. of Computer and Information Science,
Linköping University, Sweden

E-mail: {petar,petfr}@ida.liu.se

Abstract

Java is becoming an important programming language for parallel and distributed computing. Large scientific problems are increasingly being implemented using modern programming languages such as Java. One application area is simulation code for hardware in the loop simulations, written in Java. We have designed an automatic parallelization tool that at compile time can schedule a program written in a subset of Java to parallel code to be executed on a multi-processor architecture. In the current implementation we consider Parallelization of Java methods with side-effect free, sequential code. Such code often exist in numerical simulation code. This paper gives an introduction to our work within this area, and identifies problems with static scheduling of such Java programs.

Keywords: Java, Multi-processors, Scheduling, Simulation

.

1. Acknowledgments

This work is performed in the JOSES project, supported by EC in the IST program.

2. Introduction

Java is a modern object oriented language that is becoming wide spread not only in general software development areas, but also in embedded systems and parallel and distributed computing. There are also several attempts to use Java for high performance computing [1]. Another emerging area for Java is software in embedded systems.

Embedded systems requires that the software can be optimized to fully exploit the capacity of the limited hardware. Therefore, there is a great need for new and innovative optimization methods, both for optimizing speed of the program, as well as memory consumption. For computationally heavy programs with short deadlines in an embedded systems, there is also a need for parallel execution, using a multi-processor architecture. A typical application can be hardware in the loop simulations, where an embedded systems sometimes is a requirement. Such applications are studied in another EC project, called RealSim, where research within the field of real-time simulation for multi physics is conducted [3, 2]. An industrial application from ABB, used in the Realsim project, is hardware-in-the-loop simulation of power electronics in an electrical train, with extremely hard real time requirements, with response times down to 40 us.

To be able to meet hard real time requirements, often found in embedded systems, automatic parallelization of Java programs is needed. In this paper we explore the possibilities of building such a tool for a subset of Java. Our tool should parallelize a Java method, containing side-effect free, sequential code. Such code is commonly used in simulation code for hardware-in-the-loop simulations.

The automatic parallelization tool parses a Java program and produces a task graph, $G = (V, E)$. A task $v \in V$ is defined as an arithmetic operation, or a function call, etc. Each Edge ($e \in E$) imposes a data dependency on the tasks. For instance, the following code:

```
float calc(float a, float b,
           float c, float d)
{
   Exception e;
   float x,y,z;
   try {
     x = b*d + foo(a*c);
   } catch (Exception e) {
     dump_errormsg();
     exit();
   }
   y = bar(fie(c+d),a);
   if (y==0) {
      z = foo(a);
    } else {
      z = fie(x);
    }
return z;
}
```

will produce the graph depicted in figure 1. Nodes for representing if statements, i.e. control flow, are collected as one task, but we still represent the internal structure (also using a data dependency graph) of the if statement. The example also shows that we disregard exception handling, even though this is an important feature of the Java programming language. However, this can be added later in the implementation. There are some issues to deal with when allowing exceptions. For instance, can we throw an exception on one processor, and catch it on another processor.

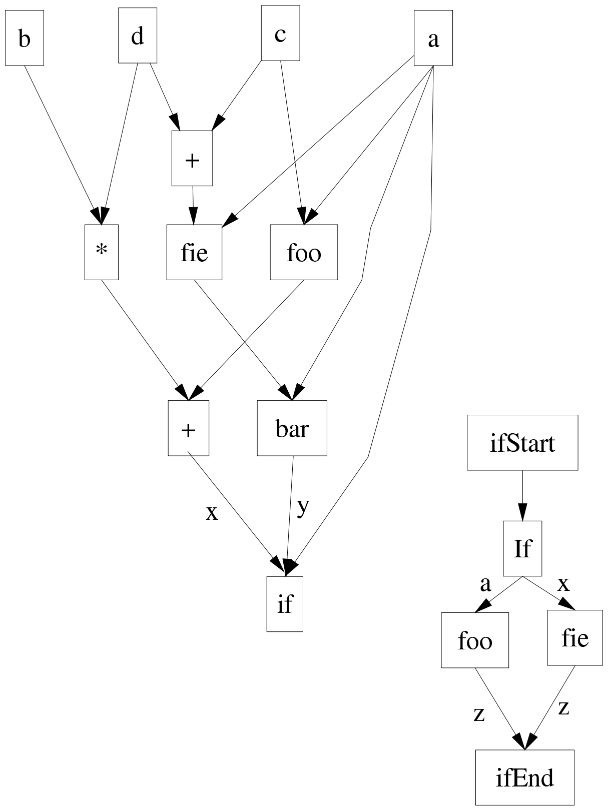


Figure 1. An example graph of a DAG.

A scheduling algorithm also needs an execution cost $e(v)$ for each node $v \in V$ and a communication cost $c(e)$ for each node $e \in E$.

3. Related Work

A large number of static and dynamic scheduling and partitioning algorithms for scheduling of task graphs to a multiprocessor system are presented in the literature. Some of them use a static technique called list scheduling [4, 7, 10, 6, 5, 9]. A list scheduler keeps a list of tasks that are ready to be scheduled, i.e. all its predecessors have already been scheduled. It selects one of the tasks in the list, by some heuristic, and assigns it to a suitable processor.

Another technique is called critical path scheduling [11]. The critical path of a DAG is the path having the largest sum of communication and execution cost. The algorithm calculates the critical path, extracts it from the DAG and assigns it to a processor. After this operation, a new critical path is found in the remaining DAG, which is then scheduled to the next processor, and so on. One property of critical path scheduling algorithms is that the number of available processors is unbounded, because of the nature of the algorithm.

A third class of scheduling algorithms is the task duplication scheduling algorithms [11][8]. These algorithms rely on task duplication as a means of reducing communication cost. However, the decision if a task should be duplicated or not introduces additional complexity to the algorithm, pushing the complexity up in the range $O(v^3)$ until $O(v^4)$.

4. Scheduling of Java Programs

After parsing Java programs (or programs written in a subset of Java) and building the task graph(DAG), we need to add costs to the graph. Each node $v \in V$ is assigned an execution cost. This means that we must be able to estimate the execution time for each expression in the Java program. For arithmetic expressions, this is trivial, but for functions and loops, it can be a bit more complicated.

The execution cost for a function can be estimated by summarize the execution cost of all statements in the body of the function, and adding a constant execution cost for the actual calling of the function. Recursive functions are not considered in our tool, i.e. they are not part of the subset of Java that we are targeting.

The execution cost of a loop can be easily calculated if the loop iterator is known. However, if the loop iterator is unknown at compile time, we need to estimate the execution cost of the loop. Here we can use several approaches. First, we can allow annotations to the Java program, informing the compiler of the maximum number of iterations for each loop having an unknown iterator. Another approach, suitable in for instance simulation code, is to use profiling information to determine the execution cost for a node. By executing the program and measure the time each loop takes, we can use that information as an approximation on each loops execution cost.

The scheduling algorithm also needs a communication cost associated with each edge $e = (v_1, v_2) \in E$. This cost is proportional to the communication time for sending a message from task v_1 to task v_2 if v_1 and v_2 are scheduled on different processors. If they are scheduled on the same processor, the cost is assumed to be zero. As a simplified model, we can have a cost proportional to the amount of data to send.

We have deeply investigated an algorithm for scheduling of task graphs called TDS. It is a critical path scheduling algorithm with task duplication, that produces optimal results, given some constraints on the graph. This constraint relates the communication costs and the execution costs of join nodes, i.e. nodes having more that one predecessor. And the optimality constraint is fulfilled if the task graph is coarse grained, The complexity of the TDS algorithm is $O(v^2)$, where v is the number of tasks. The TDS algorithm can not guarantee a fixed number of processors used for a particular problem. Early implementations of our tool indicated that such a scheduling algorithm needs a second phase for limiting the number of processors to a fixed number.

One problem with the TDS algorithm is that it can only guarantee optimality given a certain constraint [11]. This constraint is somewhat related to the granularity of the task graph. Therefore, for a practical use of the TDS algorithm, it is necessary to cluster tasks into sufficient sized grains (several tasks) prior to scheduling.

Another problem with the TDS algorithm is that it can not guarantee a limit on the number of processors needed for a specific problem. This is due to the nature of the algorithm, where in each iteration the critical path of the graph of non-scheduled nodes is extracted and assigned to a processor. One solution to this problem is to have a second phase after the TDS algorithm that merges task lists, thus reducing the number of needed processors. This is repeated until the actual number of physical processors is met.

The strength of the TDS algorithm is the low complexity, combined with the optimality. However, if the task is fine grained or if the amount of physical processors is exceeded for a given task graph, the optimality is no longer guaranteed. In these cases, we believe that the TDS algorithm combined with a pre clustering phase that increase the granularity of the task graph can produce good results, compared to other scheduling algorithms of higher complexity.

5. Implementation

We have implemented a framework for an automatic parallelization tool that currently parses c-code with additional macros, specific for simulation code. This is similar to the subset of Java mentioned earlier in this paper. However, indications from early results showed that the two problems discussed in section 4, namely grain size and number of processors needed, have to be addressed.

The number of processors problem has been solved in the current implementation by merging tasks lists. A selection criteria is needed for selecting two task lists (processors) for merging. Possible criterias for merging are load balancing, and communication cost.

One naive approach is to merge the two lists (processors) that have the maximum communication cost between each other. This approach does not guarantee any load balancing, and might give poor result.

Another approach could be to only look at load balancing when merging task lists. This can however produce too much communication overhead.

A third approach is to merge the two task lists (processors) with the earliest finish time, thus ensuring that the execution time of the program will not increase. This will of course only be true if the total execution time is larger than the execution time for the merged task list.

6. Future Work

Ongoing work includes adapting the current framework for Java code. This means that the tool should read Java programs, and generate Java programs using MPI, like for instance JavaMPI [1].

The current investigations and adaptions made on our parallelization tool are suitable for simulation code, containing a large number of simple arithmetic expressions, which makes the granularity of the task graph for such code small, i.e the tasks are small compared to the communication cost. This may not be the case for numeric computational sequential code, written in Java. For Java programs, method calls in the sequential code is more frequently represented. Thus, the task granularity is more spread. This can for instance affect the choice of which pre clustering approach is taken, and also how to merge.

Future extensions of the automatic parallelization tool is to introduce dynamic scheduling, to solve problems where the task execution and communication costs can not be determined at compile time. Dynamic scheduling can also be more useful when we have applications that contain a large proportion of control dependencies.

References

[1] http://www.javagrande.org/.

[2] *The Modelica Language, http://www.modelica.org.*

[3] The RealSim project, http://www.ida.liu.se/ pelab/realsim/index.php3.

[4] Andrei Radulescu, Arjan J.C van Gemund. FLB:Fast Load Balancing for Distributed-Memory Machines. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, 1999.

[5] C. Hanen, A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small

communication delays. Technical report, Laboratoire Informatique Theorique Et Programmation, Institut Blaise Pascal, Universite P.et M. Curie, 1999.

[6] C.Y. Lee, J.J. Hwang, Y.c. Chow, F.D Anger. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, 7(3), 1988.

[7] Gilbert C. Sih and Edward A. Lee. Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[8] G.L. Park, B. Shirazi, J. Marquis. DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. In *Proceedings of Parallel Processing Symposium, 1997*.

[9] J.J. HHwang, Y.C. Chow, F.D. Anger, C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *Journal on Computing*, 1989.

[10] Min-You Wu, Daniel D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, 1, 1990.

[11] Sekhar Darbha, Dahrma P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *Transactions on Parallel and Distributed Systems*, 9(1), 1998.

Trade-offs in Symbolic Cost Estimation of Parallel Programs

Arjan J.C. van Gemund, Hasyim Gautama

Dept. of Information Technology and Systems
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
email: a.j.c.vangemund@its.tudelft.nl

Abstract

Analytic cost estimation is a valuable aid to assess the effect of various machine and mapping parameters on program performance. Cost estimation is either based on a model of the parallel algorithm or on a model of the actually generated machine code. Especially in case of a distributed-memory system the difference in abstraction is large. In this paper we study the trade-off between analytic cost estimation at high (program) level and low (machine) level. We show that, despite its high abstraction, program level cost estimation provides the best prediction quality. This approach is implemented in a cost estimation engine within a compiler for Spar, a parallel Java dialect.

1 Introduction

In parallel/distributed embedded systems programming performance prediction plays an important role, either to predict the effects of a particular design choice, or to diagnose the cause for some observed performance behavior. Especially in the case of distributed-memory architectures mapping the program to the machine includes task and data placement which significantly complicates the mapping problem.

Symbolic performance modeling is an analytical technique concerned with mapping a program into an explicit, algebraic expression in terms of the program and machine parameters of interest, that typically predicts the execution time. An important benefit of symbolic performance models is that they offer *analytic* information regarding the complex interplay between the various program and machine parameters involved (e.g., problem size, task/data partitioning parameters, number of processing and/or communication resources, interconnection parameters). Another important benefit of symbolic performance modeling is that the resulting model has minimum solution *cost*, both in space and time. Especially in view of the large problem and/or system dimensions involved in embedded parallel processing, ultra-low prediction cost is essential for performance models to be of interest in interactive performance evaluation and/or optimization. The trade-off, compared to alternative estimation techniques based on queuing models, Petri nets, and Markov models, and, of course, simulation, is that the symbolic models generally have a limited accuracy. However, when applied at the first stages of program design where a user is more interested in quickly finding his/her way in the multi-parameter design space than in percent level prediction accuracy, symbolic cost estimation is the preferred option.

A fundamental issue in parallel program performance prediction is at what intermediate level program modeling should be performed. We distinguish between two modeling levels, namely the *program* level and the *machine* level. The program level, i.e., the level at which the (intermediate) program is still expressed in terms of the original source level, is attractive because at this level the original control flow parallelism is still visible (allowing better analysis [3]), while at the same time, there are better chances for compiling cost estimates that have a reasonable source level interpretation. At the same time, however, many aspects relating to the behavior of lower level transformation (optimization) engines must be predicted which is inherently difficult. On the other hand, the machine level, i.e., the level at which the code has already been transformed in terms of the SPMD message-passing machine interface, offers the opportunity of deriving cost estimates of the actually generated code, potentially gaining prediction accuracy. A serious drawback, however, is that symbolic cost estimation is not amenable to the message-passing paradigm. Consequently, the gain in prediction accuracy cannot be realized. Moreover, due to the various code transformations, cost estimation with any form of source level interpretation is practically infeasible [8].

In this paper we study the trade-off between performance modeling at program and machine-level in the context of symbolic cost estimation for message-passing architectures. More specifically, we demonstrate that the choice for machine-level modeling inhibits the use of symbolic prediction techniques. We also show that, despite the abstraction, program-level symbolic predictions can indeed be derived that account for the most significant performance effects that occur in the actual message-passing SPMD code. At first glance, this result may seem somewhat counter-intuitive. Yet it underlines the great potential of program-level prediction, provided the right symbolic cost modeling methodology is used.

In Section 2 we present this modeling methodology, which is based on a concept called *contention modeling*. In Section 3 we demonstrate the effectiveness of this methodology when applied at program level in contrast to machine level cost estimation. In Section 4 we describe an implementation of our methodology in a parallel Java compiler. In Section 5 we summarize our work.

2 Symbolic Cost Estimation

In this section we introduce our cost modeling approach which is based on the PAMELA (PerformAnce ModEling LAnguage) methodology [5]. More specifically, we focus on the PAMELA subset for which symbolic, closed-form cost estimations can be derived. Though the formalism and associated cost estimation technique is described at length elsewhere [4], for convenience of reference we will briefly summarize the technique in Section 2.1. In Section 2.2 we describe our specific modeling approach called "contention modeling". In Section 2.3 we present the rationale behind our approach.

2.1 Modeling Algebra

The language subset we consider comprises a process algebra based on sequential, parallel, and conditional composition operators. The algebra features binary infix operators to describe sequential composition (';'), and fork/join-style parallel composition ('||'). Sequential and parallel replication are expressed by **seq** and **par** prefix operators, defined by **seq** $(i = a, b)\ L_i = L_a\ ;\ \ldots\ ;\ L_b$ and, similarly, **par** $(i = a, b)\ L_i = L_a \parallel \ldots \parallel L_b$. The algebra also features **if** and **while** operators which are defined in the usual way.

While the *condition synchronization* [2] (CS for short) provided by the above operators allows for the expression of any series-parallel (SP) computation structure, *mutual exclusion* [2]

(ME for short[1]) can be specified by the **use** construct, like in **use**(r, τ) where the invoking process exclusively acquires resource r (FIFO without preemption, non-deterministic conflict resolution) for τ *time* (excluding possible queuing delay). In fact, any time delay associated with spending cycles (i.e., work) in a computation is expressed (i.e., charged to some resource) by **use** statements. A resource s can have a multiplicity, denoted $|s|$, that may be larger than 1. Like in queuing networks, it is convenient to define a resource ρ such that $|\rho| = \infty$, usually called infinite server. Instead of **use**(ρ, τ) we will simply write **delay**(τ). Unlike the **use** operation, a **delay** operation will never entail additional queuing delay on top of its programmed delay.

As a simple example, consider the PAMELA model (by convention denoted L), of some parallel computation described by the following process expression $L =$ **par** $(p = 1, P)$ **use**$(cpu_{f(p)}, \tau_p)$ where cpu_i denotes processing resource number i and τ_p models its workload. If $cpu_{f(p)}$ is unique (e.g., a mapping $f(p) = p$), L represents a time delay equal to $T = \tau_1 \max \ldots \max \tau_p$, as each **use** statement runs in parallel. In contrast, however, if $f(p) = c$ (i.e., constant, each process is mapped to the same CPU), it follows $T = \tau_1 + \ldots + \tau_p$, as a result of the serialization of the N parallel requests due to ME. Although PAMELA features more operators our approach to modeling parallel computation will be essentially expressed in terms of **par**, **seq**, and **use**.

A PAMELA model L can be directly executed which corresponds to performance simulation. However, the highly structured operators for both CS (**par**, **seq**) and ME (**use**) enable a simple, compile-time analysis technique where PAMELA models are *compiled* into symbolic cost models through a completely mechanical procedure. Due to the non-determinism that arises with ME, the time cost of computations in which ME plays a role is typically stochastic. Aimed to provide a low-cost, analytic (i.e., deterministic) estimate, the analysis algorithm computes a lower bound which in some cases entails a prediction error. The specific advantage of the estimation algorithm compared to conventional techniques (static analysis, complexity analysis), however, is that the estimation error of T is bounded. Theory and experiments show that the average estimation error due to the synchronization effects is less than a constant factor 2, *regardless* of the type of parallel computation (as long as it can be expressed in terms of our process algebra), while in most cases the average error is well within tens of percents. Although applied in the sequel, due to space considerations the cost estimation algorithm itself is not described in the paper. Details can be found in [4].

2.2 Contention Modeling

A classical example in performance modeling is the machine repair model (MRM) [9] in which P clients either spend a mean time τ_l on local processing, or request service from a server s ($s = 1$), with mean service time τ_s, for a total cycle count of N iterations. Both time delays are typically stochastic. The PAMELA model of the MRM is given by

$$L = \textbf{par}\ (p = 1, P)\ \textbf{seq}\ (i = 1, N)\ \{\textbf{delay}(\tau_l)\ ;\ \textbf{use}(s, \tau_s)\}$$

in which the exclusive service is expressed by the **use** operation applied to the passive resource s that represents the server. Note that in our modeling approach the server is a passive *resource*. The ME arising from sharing the resource is modeled in terms of *contention*, rather than communication. (In a message-oriented approach s is modeled by a reactive *process*.) Hence, we

[1]CS represents the static form of process synchronization, while ME represents dynamic process synchronization ("contention").

have coined our modeling approach *contention modeling*[2]. We will discuss the disadvantage of the alternative, message-oriented approach later on.

Despite the advantages in analytical sense as described earlier, it would seem that the constraints imposed by the highly structured synchronization operators entail a drastic reduction in modeling power. For example, the SP restriction with respect to CS would make it impossible to model a fundamental parallel computation schedule such as *pipelining*. However, the use of contention modeling does allow pipelining to be expressed in terms of our cost modeling framework. Consider a pipelined computation involving N data sets processed by an S stage pipeline (e.g., vector unit, packet-switched communication pipeline, software pipeline). In a message-oriented paradigm each pipeline unit would map to a process that would synchronously receive a data set, process it, and send it to the next unit. In contention modeling, however, the entire computational process is expressed for each data set. Each data process is executed in parallel and *contends* for each unit in the course of its propagation through the pipeline. The PAMELA model is given by

$$L = \textbf{par}\ (i = 1, N)\ \textbf{seq}\ (s = 1, S)\ \textbf{use}(u_s, \tau_c)$$

where u_s denotes the resource corresponding to stage s, and τ_c denotes the associated processing time (cycle time) per unit. The above model correctly accounts for both startup delay as well as the bandwidth of the pipeline. Note that, while the absolute order in which data is processed is left undetermined, the *time cost* prediction is always valid. Application of the earlier estimation algorithm (including optimizations [5]) yields the exact result $T = (S + N - 1)\tau_c$.

Notice that in our contention modeling paradigm the pipeline can be conveniently expressed as an SP model based on the use of ME, whereas a usual description of all the task precedences merely in terms of CS would necessitate a non-SP description (which is not amenable to our cost compilation algorithm). Hence, the example is a typical illustration of the modeling power of the contention modeling technique. It should be stressed that contention modeling is not some contrived way of describing parallel computation. It essentially expresses (and preserves) the *potential* parallelism that exists within the *algorithm* independent of the actual machine *implementation*, where ME models the specific resource limitations. This separation of algorithm and machine offers a *portable* way of modeling.

2.3 Rationale

The choice for the high-level, contention-oriented modeling paradigm in PAMELA is motivated by the fact that message-oriented models are not amenable to symbolic cost estimation. This is why symbolic cost modeling is best applied at program level where the programming model is still a shared-memory model instead of at the message-passing machine level. We illustrate the fundamental problems involved with analyzing message-passing models with a simple example. Recall the MRM. In a message-oriented paradigm, both clients and server would map to processes that communicate (and synchronize) using message-passing constructs. Consider a message-oriented version of PAMELA based on the use of a CSP-like scheme [6] using synchronous **send** and **receive** operators combined with a selective communication construct ('□') to achieve scheduling non-determinism. Let L_p, $p = 1, \ldots, P$ denote the P client processes and let S denote the server. The MRM is then modeled by the following set of process equations

[2]The original terminology *material*-oriented modeling, and its dual, *machine*-oriented modeling, stem from the domain of simulation of, e.g., plant production lines [7]. We feel that the application of the material-oriented paradigm in the specific domain of parallel programming justifies using the distinct name "contention modeling".

```
$L =$  $S \parallel \{\textbf{par}\ (p = 1, P)\ \textbf{seq}\ (i = 1, N)\ \{\textbf{delay}(\tau_l)\ ;\ \textbf{send}(S)\ ;\ \textbf{receive}(S)\}\ \}$
$S =$  $\textbf{while}\ (\textbf{true})\ \{$
           $\textbf{receive}(L_1)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_1)\ \Box$
           $\textbf{receive}(L_2)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_2)\ \Box$
              $\ldots$
           $\textbf{receive}(L_P)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_P)$
       $\}$
```

By the way, note that in this model the mutual exclusion at the server (implicitly) results from the single thread of control within S while the required non-determinism results from the '$\Box$' operator.

In contrast to the contention model, for the above CSP-type solution there exists no general, automatic analysis scheme that produces a useful symbolic cost estimate. Especially when message-passing models become complex, it is impossible to efficiently deduce the critical synchronization path that now actually runs across multiple processes. The problem is due to the explicit use of a non-deterministic operator ($\Box$), in combination with providing *separate* constructs for sending and receiving which might lead to non-SP structures. In contention modeling, CS and ME are kept orthogonal, both in terms of *single*, high-level operations. Consequently, the need to first "reverse engineer" to a higher level model (impossible in a mechanized scheme) is completely avoided.

This property of the message-oriented modeling paradigm implies that program code expressed in terms of a message-passing architecture cannot be symbolically analyzed in terms of closed-form expressions. This is essentially the rationale for choosing a program-level contention modeling approach for deriving program optimization logic. Another striking example of the problems associated with analyzing message-passing code is described in the following section.

3 Case Study

In this section we will study a simple distributed-memory application kernel in order to demonstrate the prediction problems associated with the choice of a machine level as the basis for performance feedback. Consider the simplified line relaxation algorithm fragment [1] applied to an $N \times N$ matrix A in which the relaxation sweep direction is in the i direction (see Fig. 1). In

```
for i = 1 ..  N-2
   forall j = 0 ..  N-1
      A[i,j] = A[i-1,j] + A[i+1,j];
```

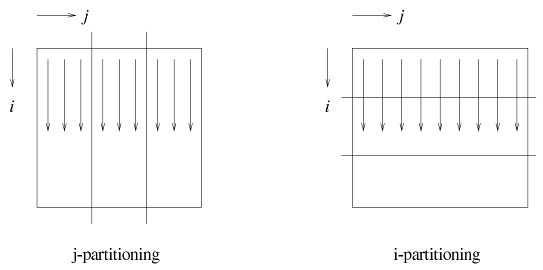


Figure 1: ADI algorithm and data partitioning choices.

the parallelization for a P processor distributed-memory machine we shall consider the choice between two regular block partitioning strategies, i.e., either on the j axis or along the i axis (a choice, by the way, that is clearly trivial from a human point of view). The j axis partitioning does not introduce any problems and the machine level code clearly reveals the $O(P)$ speedup involved. The i axis partitioning, however, introduces prediction problems when modeled at machine level.

The SPMD code in the i partitioning case is characterized by the following pseudo code (following the usual "owner-computes" convention including a number of trivial index optimizations), where p denotes the processor index and L and U denote the processor-specific index bounds ($U(p) - L(p) = \mathcal{O}(N/P)$). Note that the original (shared) data indexing is used for readability.

```
if p > 0
   send(p-1,A[L(p),:]);
if p > 0                    // get A[L(p)-1][:]
   recv(p-1,tmp_l);
else
   tmp_l = A[0,:];
if p < P                    // get A[U(p)+1][:]
   recv(p+1,tmp_u);
else
   tmp_u = A[N-1,:];
for i = L(p) .. U(p) {      // update partition
   if i = L(p)
      for j = 0 .. N-1
         A[i,j] = tmp_l[j]+A[i+1,j];
   if i > L(p) and i < U(p)
      for j = 0 .. N-1
         A[i,j] = A[i-1,j]+A[i+1,j];
   if i = U(p)
      for j = 0 .. N-1
         A[i,j] = A[i-1,j]+tmp_u[j];
}
if p < P
   send(p+1,A[U(p),:]);
```

In this example, we will ignore the fact that the j loops can be vectorized.

The above SPMD code illustrates the difficulties involved when compiling generated SPMD code into a symbolic performance model. Although the local bounds on the i loop are reduced by a factor P (as in the j-partitioning), and therefore suggest $O(P)$ speedup, in reality the SPMD code has no speedup at all. This is due to the communication scheme, which totally *serializes* the entire computation (all but the first processor initially block at the `recv` statement). As discussed in Section 2.3, however, especially in a symbolic analysis it is generally hard to deduce that the the critical path now runs through each individual SPMD process (note that the serialization phenomenon is even hard to notice for humans).

Clearly, the critical path that is obscured by the above message-passing implementation is nothing but the result of the explicit sequential i loop at program level. Consequently, it is much more attractive to consider a modeling approach at *algorithm* level (program level) than at *implementation* level (machine level). In order to abstract from the actual partitioning implementation (either for shared-memory or distributed-memory systems) we model the original computation with its full (potential) parallelism while each mapping decision is expressed in terms of a *contention* model. In fact, we will used the same contention modeling approach as in the vectorization example where the potential parallelism of the vector operation is expressed while the machine resources determine the actual parallelism.

The performance model of the algorithm is then expressed according to

$$L_{ADI} = \textbf{seq}\ (i = 1, N-2)\ \textbf{par}\ (j = 0, N-1)\ \textit{update}(i,j)$$

where *update*(i,j) denotes the update of element A_{ij}. Let the mapping function $\mu(i,j)$ denote the processor resource responsible for the update of A_{ij}. Then the model of *update* is given by

$$\textit{update}(i,j) = \textbf{use}(\textit{cpu}_{\mu(i,j)}, \tau_f)$$

where τ_f denotes the computation time associated with the update of A_{ij} (in this example we ignore communication delays for simplicity). Note, that this represents a "processor contention model".

Based on this model, the evaluation of both partitionings is straightforward. For the i axis partitioning it holds $\mu(i,j) = i/B$. Consequently

$$L_{ADI} = \textbf{seq}\ (i = 1, N-2)\ \textbf{par}\ (j = 0, N-1)\ \textbf{use}(\textit{cpu}_{i/B}, \tau_f)$$

which, in contrast to the actual SPMD code, directly reveals the algorithm's sequential nature *and* the absence of speedup. Indeed, L directly compiles to $T = (N-2)N\tau_f$. Thus by exploiting the algorithm's inherent sequential semantics present in its original description, from a simple cost estimation it directly follows that in the present algorithmic form an i axis partitioning will not yield any speedup. In contrast to the SPMD implementation, the inherent sequentialism is easily detectable. Clearly, at this point the potential accuracy benefit from modeling at machine level is hardly relevant.

The line relaxation case study illustrates the advantage of the contention modeling approach when applied at program level in order to express compile-time (i.e., symbolic) optimizations. Although the approach cannot deal with message-passing programs, the information contained at the high, shared-variable level captures the high data mapping sensitivity of the application, which is totally blurred at the lower, message-passing level anyway. Of course, there are cases where the choice to abstract from the actual message-passing implementation may induce a considerable error as the low-level code generation model is not included in the cost model. For instance, a naive message-passing code generation model could completely sequentialize an inherently parallel operation[3]. Clearly, the above prediction method will not take this into account (although inherent sequentializations at *algorithmic* level would still be accounted for, of course). Note, however, that the focus of our approach is towards predicting the effects of user-level code and/or data mapping decisions, rather than capturing performance idiosyncrasies of an underlying compilation model that does not properly implement a source program of which the code and data mapping degrees of freedom have already been determined by a user. A mature code generation model must therefore be assumed, e.g., which does not introduce such pathological schedules as in the above example (note that a simple inspector/executor implementation would already solve the problem just mentioned in the footnote).

[3]For example, consider a simple computation $y_i = f(x_{i-1})$ where each element of x and y are mapped onto a separate processor (x and y aligned). A naive, scalar "owner-computes" scheme in which each processor (sequentially) traverses the entire index space in the positive i direction will completely (and unnecessarily) sequentialize the computation.

4 Implementation

Our program level cost estimation approach is being implemented in the cost estimator engine (or CE for short) which is integrated in the Delft University Spar compiler. Spar [12, 11] is a Java derivative that extends Java with language and annotation (pragma) constructs to enable parallel computing on both shared-memory and distributed-memory architectures.

The CE interfaces to the Spar compiler in the following way. The Spar compiler front-end parses a Spar file, producing an intermediate representation expressed in terms of the intermediate language Vnus [10]. The CE interfaces at the Vnus level, before the transformation towards the SPMD message-passing machine interface has been performed. This essentially reflects our program level approach. The cost estimation approach is depicted in Figure 2. The CE engine

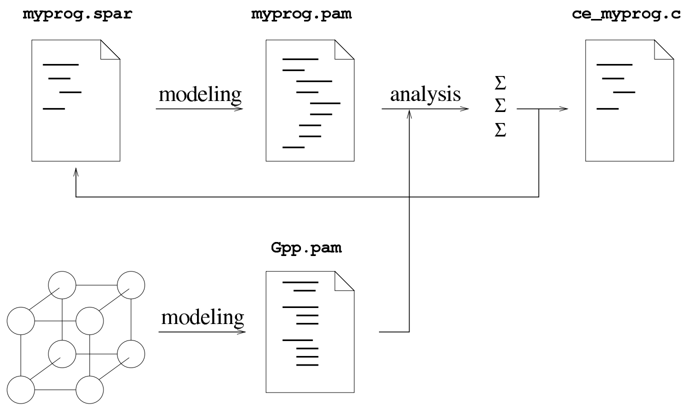


Figure 2: CE Approach

can be divided into the following two modules:

- A modeling module in which a Spar program (`myprog.spar`) is transformed into a Pamela* model (`myprog.pam`). Based on the `processor` type pragma in the Spar program that designates the specific Pamela* machine model(s) to be used, the program model contains an `include` statement referring to the corresponding machine model (`Gpp.pam`), effectively yielding a complete Pamela* model of the application. The machine model is to be supplied by the user according to a pre-specified interface format. As the Pamela* language has been chosen as interface formalism between the application and the actual CE analysis engine, machine model coding is performed in the Pamela* language itself, which makes coding straightforward [5].

- An analysis module in which the Pamela* model is transformed into a symbolic cost model [3]. The analysis is performed by a Pamela* compiler that has been developed as a stand-alone tool. The application cost model is exported to a parameterized C function (as in the figure), and/or a Maple file to allow the user to perform interactive parameter (e.g., mapping or scalability) studies.

5 Conclusion

In this paper we have studied the trade-off between modeling at program level and machine level in the context of symbolic cost estimation for message-passing architectures. Despite the abstractions involved, we have shown that program level modeling is to be preferred for symbolic cost estimation, provided this method is used in conjunction with a modeling technique called "contention modeling". We outlined this modeling technique in terms of the PAMELA language, which is specifically tailored to this way of modeling, and which automatically produces low-cost, symbolic time cost expressions to guide the algorithm design and mapping process.

Acknowldgement

Part of this work has been performed within the ESPRIT LTR Project 28198 "JOSES" (Java and CoSy Technology for Embedded Systems).

References

[1] V.S. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren and C-W. Tseng, "Requirements for data-parallel programming environments," *IEEE Parallel and Distributed Technology*, Fall 1994, pp. 48–58.

[2] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no. 24, 1983, pp. 132–145.

[3] A.J.C. van Gemund, "Compiling performance models from parallel programs," in *Proc. 8th ACM Int'l Conf. on Supercomputing*, Manchester, July 1994, pp. 303–312.

[4] A.J.C. van Gemund, "Compile-time performance prediction of parallel systems," in *Computer Performance Evaluation: Modelling Techniques and Tools* (H. Beilner and F. Bause, eds.), LNCS 977, Berlin, Springer-Verlag, Sept. 1995, pp. 299–313.

[5] A.J.C. van Gemund, *Performance Modeling of Parallel Systems.* PhD thesis, Delft University of Technology, The Netherlands, Apr. 1996.

[6] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, Aug. 1978, pp. 666–677.

[7] W. Kreutzer, *System simulation, programming styles and languages.* Addison-Wesley, 1986.

[8] F. Kuijlman, "Vnus mapping engine." Personal Communication.

[9] S.S. Lavenberg, *Computer Performance Modeling Handbook.* New York: Academic Press, 1983. ISBN 0-12-438720-9.

[10] C. van Reeuwijk, "Vnus language specification 2.1 beta 1," tech. rep., Delft University of Technology, Delft, The Netherlands, Apr. 2000.

[11] C. van Reeuwijk, F. Kuijlman, H.J. Sips and S.V. Niemeijer, "Data-parallel programming in Spar/Java," in *Proc. of the Second Annual Workshop on Java for High-Performance Computing (in conjunction with 2000 ACM ICS)*, May 2000, pp. 51–66.

[12] C. van Reeuwijk, A.J.C. van Gemund and H.J. Sips, "Spar: A programming language for semi-automatic compilation of parallel programs," *Concurrency: Practice and Experience*, vol. 9, Nov. 1997, pp. 1193–1205.

Session III

Boosting Performance

The JOSES Project

Compiling Java for Embedded Systems

Arthur Veen, Marcel Beemster, Rob Kurver and Hans van Someren

ACE Associated Compiler Experts bv
Van Eeghenstraat 100
1071 GL Amsterdam
The Netherlands
arthur@parcom.nl, {marcel,rob,hvs}@ace.nl

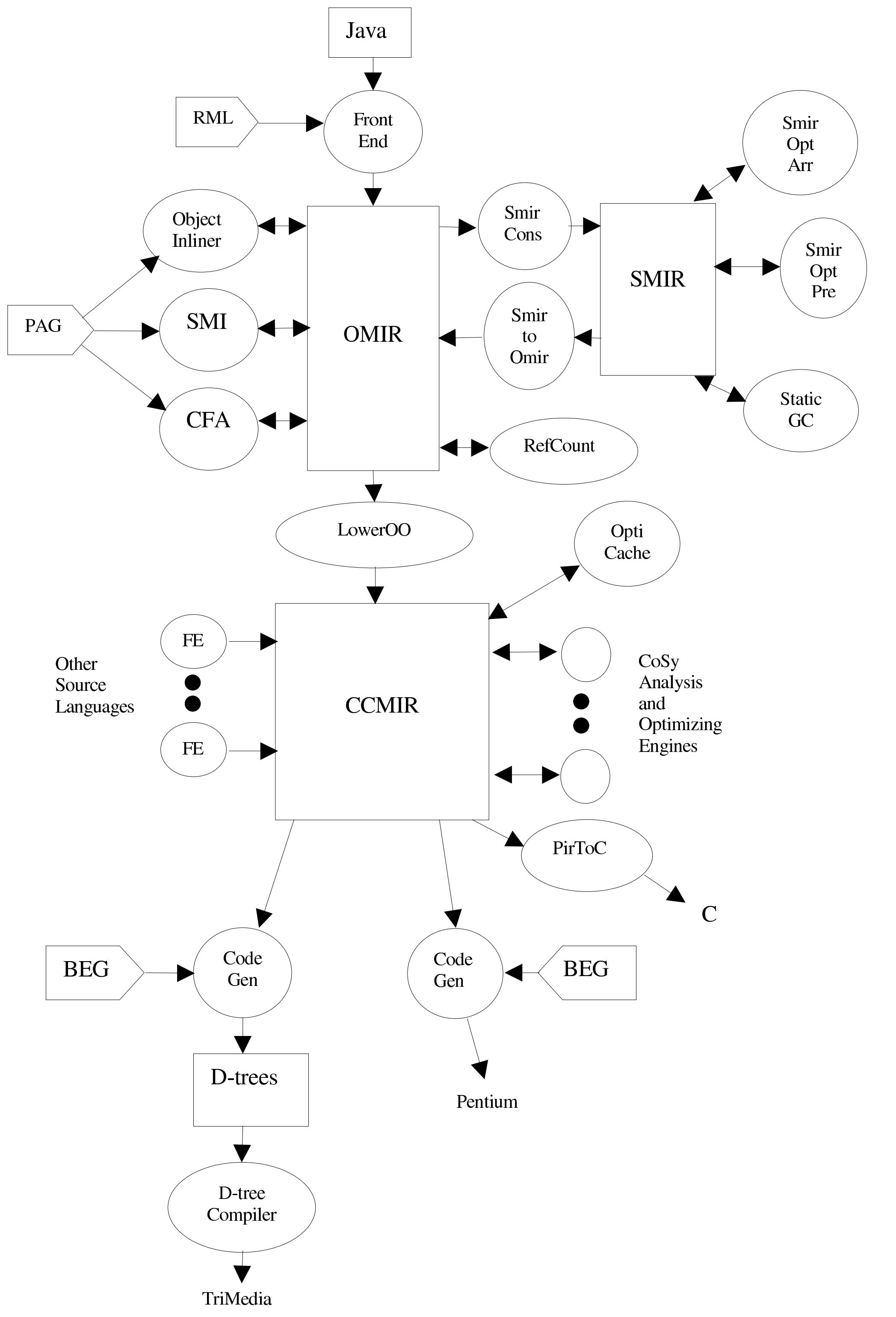
Summary

The high memory consumption and the low speed of current Java implementations have prevented the wide-spread usage of Java for embedded systems. In the JOSES research project a highly optimizing Java compiler is implemented to overcome these problems. Optimizations have been designed to deal with the overhead of dynamic method invocation, object representation, pointer de-referencing and automatic memory management. Special attention is paid to effective usage of the data cache.

1 Introduction

Java is a powerful and sophisticated language that offers many software engineering advantages. Some of these advantages, ease of maintenance, re-use of components and portability, are especially important for the embedded systems market, because of the rapid succession of hardware components. However, Java has hardly been used in this market segment, because current Java implementations do not provide sufficient speed or consume too much memory. The JOSES project investigates whether sophisticated compilation techniques can solve this problem, i.e. whether a state-of-the-art Java compiler can generate code that is efficient enough in size and speed to make the usage of Java for embedded systems attractive.
The JOSES project is an example of fruitful cooperation between academic and commercial partners. The object-oriented features of Java require the development of new and complicated optimization techniques. For such developments the freedom and stimulation of the academic world is most suitable. On the other hand the tuning and realistic evaluation of such techniques require access to an efficient and flexible compilation system. A consortium was therefore formed consisting of Europe's leading compiler technology manufacturer, ACE Associated Compiler Experts, and some of the best academic compiling groups in Europe: Universität des Saarlandes, Universität Karlsruhe, Linköping University and Technical University Delft. To keep the researchers focussed on commercially relevant optimizations two companies, Philips and Ericsson, were added to the consortium in the role of end-user.
The JOSES project has been funded as an ESPRIT long term research project by the European Union (EP 28198). It started at the beginning of 1999 and will end in the middle of 2001. About 20 people are involved and the total budget is the equivalent of 17 person years.
The JOSES project entails more than can be described in this paper. An important part of the project not covered in this paper investigates the introduction of parallelization constructs into Java and the efficient compilation of these. A sophisticated symbolic cost estimator is also under development, that generates an analytic cost model of the parallelized Java program. The cost model is to be used by the parallel programmer to assess the performance effects of machine parameters such as the number of processors, computation and communication bandwidth, etc., and to predict the effects of different code and data mapping schemes. All of this work is especially relevant for heterogeneous multiprocessors on embedded systems. Further information on this exciting work can be found in [Gemu94], [GeGa01] and [RDKS00]. The use of multiprocessors is also investigated in a project task that investigates the automatic partitioning of sequential programs into tasks [ArFr01].

2 Technical Structure



The JOSES Java compiler is being developed with ACE's proprietary CoSy compiler development system (see `//www.ace.nl` and [AAS94]). The power of CoSy is that it unifies compilers for many different source languages and many different target processors. This unification is achieved by providing an integrated set of front-ends and back-ends. Each front-end translates a particular source language to one rigidly defined common Intermediate Representation (IR). Each back-end translates the IR to code for a particular processor target. The power of the compiler is greatly enhanced by a large collection of analyzing and optimizing engines: analyzers merely deposit information into the IR, while optimizers actually transform the IR. Since these engines all work

on the IR, they are available for every combination of source language and target processor. Despite the power and complexity of the total CoSy system it is easy to add new source languages, target processors or optimizers. This is facilitated by the use of task-specific generators for the intermediate representation, the engines and their interfaces. An important task of these generators is to enforce strict modularization rules.
The figure shows the main components of the JOSES compiler. Rectangles depict formalisms (languages), ellipses depict *engines*, i.e. software components working on these formalisms, and the pentagons depict generators. The Java front-end is generated from a specification in the formalism *Natural Semantics* by the RML generator developed by the University of Linköping. There are different levels of IR. Java is first translated to an object-oriented IR, called OMIR. On this level a number of analyzing and optimizing engines work, most of them generated by the PAG Program Analyzer Generator [Mart98] developed by the Universität des Saarlandes. Certain optimizations require a Static Single Assignment view of the OMIR, called the SMIR. The *LowerOO* engine transforms the OMIR into the common IR (CCMIR). At this level the cache optimization engines work as well as all the standard analyzers and optimizers available in CoSy. Currently the JOSES compiler is tested for two target processors, SPARC and TriMedia, but other code generators can be added easily.
The generated code is linked with the standard class library from SUN (interfaced through JNI) and a small JOSES-specific run-time system. The latter provides multi-threading support and contains a special real-time garbage collector that prevents unbounded interruptions.

3 Object-Oriented Optimizers

Most of the performance and memory consumption problems of Java are due to its object-oriented nature or to its automatic memory management (garbage collector). In this section we describe the optimizers that are developed in JOSES to deal with these problems. All of these optimizations improve the performance, while most of them reduce memory consumption as well.

3.1 Static Method Invocation

Since Java is a pure object oriented language, all (non-static) method invocations are dynamically dispatched. Unfortunately, dynamic dispatching imposes direct run-time cost: the time spent in performing the dispatch and the memory for storing the data structures that are needed to perform the dispatch. There are indirect costs as well due to lost opportunities for in-lining and inter-procedural analysis. Furthermore, the object-oriented style of programming tends to distribute code that would have been a single procedure in traditional languages across multiple procedures. As a consequence, methods have small bodies which hampers traditional intra-procedural compiler optimizations. Even more important, the object-oriented style results in substantially more method invocations than the traditional, imperative style of programming. Thus method invocation is one of the major sources of inefficiency. If it can be determined statically that a specific method call will always result in the same procedure call, the method call can be resolved statically. This not only avoids the overhead of dynamic dispatching, but also creates opportunities for inter-procedural optimization.
In JOSES two alternative methods have been implemented to enable static method invocation.

- The first one starts with a conservative control flow graph constructed by using Rapid Type Analysis [TiPa00]. It then statically evaluates the program using data flow analysis and computes for each variable the object-types this variable may hold at run time. If a method call results in the same method being called for all of these object-types, the call can be replaced by a direct procedure call. This engine has been generated by the generator PAG [Mart98]. To enable the inter-procedural analysis for PAG a Call Graph Construction Engine has been implemented.
- An alternative way to determine statically the possible targets of method calls is Control Flow Analysis. CFA traces the sets of classes a variable may point to by formulating a set of constraints for each method invocation and by computing the least solution of this set. Usual CFA implementations generate conditional constraints. The CFA engine built in JOSES replaces these conditional constraints by macros that are evaluated when the system is solved. This promises to be a fast and scalable approach.

These methods are two different ways to look at the same problem. The first one starts by computing a (conservative) control flow graph and tries to eliminate edges from this graph. The second one starts with no inter-procedural edges at all and only introduces edges to really callable methods at call points.
A full description of this work can be found in [DePr01] and [Prob01].

3.2 Representation Analysis

Since Java is a pure object-oriented language, all data structures have to be represented by objects. Objects can only contain primitive data types or references to other objects. Furthermore, objects contain additional information, for instance about the class of the object. Thus, representing classical data structures (lists, trees, queues) by objects may introduce a lot of overhead, both in terms of memory consumption and runtime. Object creation is slow, and access to non-primitive components always has to follow reference chains. Therefore, one should seek for more compact representations of aggregate objects that decreases memory consumption and accelerates access to components.

Representation analysis (also known as unboxing) has been devised and successfully applied to the implementation of functional languages. In JOSES this work is extended to object-oriented languages. The optimization that is pursued is called *object in-lining*. The *ObjectInliner* is a composite engine that searches for fields of objects that are *in-linable*, i.e. that can safely be replaced by the object itself. It consists of a large number of small engines, many of which have been generated by PAG. They compute which variables in which statements may have live alias. On the basis of this it detects which object fields and which stores are candidates for in-lining. This set of candidates has to fulfill certain restrictions. By formulating this as an optimization problem (with the total number of in-lined fields as cost function) the dynamic equation solver is used to arrive at an optimal solution. If necessary objects are cloned to relax certain constraints. A full description of this work can be found in [Laud01].

3.3 Static Garbage Collection

In an object-oriented program a large fraction of the execution time is consumed by the creation and destruction of objects. Creation of an object requires the allocation of heap memory and destruction leads to the overhead associated with garbage collection. If the life-time of an object can be determined statically, the overhead of the garbage collection can be avoided by inserting an explicit de-allocation statement at the end of its life-time. This we call static garbage collection. In some cases the overhead can be further reduced by allocating the object on the stack. This is the case when the life-time of the object does not extend beyond the life time of the stack frame in which it was allocated.

The *StaticGC* engine attempts to determine the life-time of each object. If successful the end of the life-time of that object is marked and explicit de-allocation nodes are inserted in the graph. If it can also be determined that references to that object cannot escape its creation context, the allocation of the object is marked as *stack-allocatable.*

3.4 Cache Optimizations

Modern embedded processors use data caches to boost performance. The effect of data caches depends on locality properties of the executed program. Typical embedded applications, such as image or voice processing, have locality properties that could greatly benefit from the data cache. If these applications are programmed in languages such as Fortran or C, the application programmer has the possibility to influence the cache performance of his application, e.g., by choosing a specific memory layout for the crucial data sets. Hence, these languages make it rather easy to code applications in a way that improves data cache utilization. When Java is used, however, it is very difficult for the programmer to influence memory lay-out. Improving cache efficiency is therefore an important function of the optimizing Java compiler.

The purpose of the analyzers and optimizers described in this section is to increase cache efficiency by reducing either the proportion or the cost of cache misses. In JOSES the first method is used for accesses to arrays within loop nests and the second one for dynamic data structures as graphs and lists.

- The proportion of cache misses for array accesses within loops can be reduced by either changing the memory layout or by loop restructuring. Loop restructuring is complicated in Java programs due to the abundance of exception handling code that prevents reordering of statements. JOSES therefore concentrates on memory layout optimizations.

  First a set of analyzers (labeled *SmirOptArr*) working on SMIR finds
  - *regular arrays*, i.e. arrays of which the size in each dimension can be statically determined.
  - *regular loops*, i.e. loop nests with statically computable iteration sets
  - *regular access patterns*, i.e. accesses to regular arrays within regular loops with regular and statically computable index sets

  Since subsequent loop nests may have conflicting requirements for the memory layout, a cost based optimizer determines the optimal layout for each regular array across all regular loops. This reduces the overhead that would be needed to adjust the memory layout for subsequent loop nests with different optimization requirements. A set of engines (labeled *OptiCache*) working on CCMIR implements this transformation. The Local OptiCache engine proposes for each loop nest the best memory layout using techniques such as padding, transposition, cache line allocation and merging. A Global Cost Optimizer picks the optimal set of proposed optimizations, using the Cache Performance Estimator as a cost function.
- The second method to increase cache efficiency (reducing the cost of cache misses) can be achieved by either prefetching or by load sensitive scheduling. The latter is not explored in JOSES, since it requires sufficient Instruction Level Parallelism which is very scarce in object-oriented programs.

  The *SmirOptPre* engine searches for an instruction that is likely to cause a cache miss and inserts a prefetching instruction early enough in the execution path to ensure that the corresponding cache line is loaded in time. This engine works for iterative as well as for recursive traversal of dynamic data structures such as lists and graphs.

3.5 Real Time Garbage Collection

Objects of which the static garbage collection engine could not determine the life time have to be de-allocated by the dynamic garbage collector. A copying or mark-and-sweep garbage collector interrupts the execution of a program whenever there is insufficient free memory to serve an allocation statement and de-allocates all garbage by traversing the entire heap. This is not appropriate in a real-time environment since it makes the execution time of an allocation statement unpredictable. Predictable mark-and-sweep garbage collectors have been introduced, but these consume extra memory. In JOSES garbage collection based on reference counting is adopted, since it distributes the garbage collection overhead throughout the program: each creation or destruction of a reference to a garbage collected object involves maintenance of the reference count. Redundant reference count updates are removed by a peephole optimizer.

Although reference counting solves the problem of the unbounded execution time of an allocation statement, a straightforward implementation would shift the problem to the de-allocation statement due to cascading de-allocation (recursive decrement). When an object is de-allocated, the object is scanned for child references. The reference counts of its children are decreased. If the reference count of any of the children falls to zero, that child is also de-allocated. This makes the execution time of the de-allocation statement unpredictable. This problem is tackled by not recursively de-allocating the child objects, but by adding them to *to-be-freed* lists which are processed during later allocation statements.

For objects that are accessible from multiple threads the reference count maintenance code has to be protected by locks. It is currently investigated how this can be done efficiently.

A major drawback of reference counting is that it cannot remove cyclic data structures. In JOSES the application programmer is responsible for breaking cycles when a cyclic data structure has to be removed. This task is facilitated by a tool that finds all recursive classes. This tool can be extended to include shape analysis [WSR00].

Random allocation of heap may cause fragmentation of memory, which may significant increase memory consumption. In JOSES such fragmentation is reduced by maintaining a set of free lists, each containing memory regions of fixed size. Since using free lists is not suitable for large objects such as bit maps, an additional large object region is used.

4 Validation

One of the advantages of using CoSy is that most of the engines used in the compiler are existing CoSy engines that have already been validated in compilers for other languages. Only the engines specifically designed for JOSES need to be validated. The correctness of the compiler including all the new engines is checked by compiling, running and checking the output of a large number of Java test sets.

A first indication of the speed of the code generated by the JOSES compiler is obtained by running several popular benchmark suites, such as SPEC JVM98.

A thorough study of the quality of the code (both with respect to speed as well as size) is made using two large proprietary Java codes supplied by the JOSES partners Philips and Ericsson. The code from Philips is a simulator for the 8051XA processor.

The code supplied by Ericsson [Hagg01] is a Java application that implements a small part of a Radio Network Control system. The main component of the prototype application is a Base Station Controller (BSC) handling simplified call setup and release. The BSC is implemented in Java and intended to run on a soft real-time platform. It consists of approximately 5600 lines of Java code. The main work of the BSC is to handle call setup, call release and handover of a number of simultaneous mobile calls. Each call is handled by a Java thread during its lifetime. The BSC code will be appropriately instrumented so that factors such as communication cost and the cost for task switching can be factored out of the measurements. As performance target has been chosen that the JOSES compiler should produce code that is at least twice as fast as when using the Solaris HotSpot JVM.

5 Status and Planning

The different components of the compiler have been implemented in 2000 and the compiler with all its analysis and optimizing engines has been configured in the Winter of 2001. Thorough testing and evaluation by the end-user partners as well as tuning of the components is currently in progress. We expect to have benchmarking figures available in the Summer of 2001. Those components that have proven to be commercially relevant will be integrated in the CoSy product in 2002.

Acknowledgements

Christian Probst, Götz Lindenmaier, Arjan van Gemund and Tobias Ritzau made direct contributions to this paper. Many people have been involved in the JOSES project. Major contributions came from Pär Emanuelsson, Tom Rindborg and Patrik Hägglund (Ericsson SoftLab AB), Henk Schepers (Philips ASALab), Tobias Ritzau, Andrzej Bednarski, Peter Fritzson and Peter Aronsson (Linköping University), Kees van Reeuwijk, Arjan van Gemund and Henk Sips (Technische Universiteit Delft), Uwe Assmann, Götz Lindenmaier and Florian Liekweg (Universität Karlsruhe), Christian Probst, Peeter Laud and Reinhard Wilhelm (Universität des Saarlandes).

References

AAS94 M. Alt, U. Assmann, H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *5th International Conference on Compiler Construction, CC94*, Edinburgh, LCNS, Vol. 786, April 1994, pp.278-293.

ArFr01 Peter Aronsson, Peter Fritzson, *Static Scheduling of Sequential Java Programs for Multi-Processors.* submitted to JOSES Workshop, January 2001.

DePr01 Holger Dewes, Christian Probst, *Static Method Call in Java,* submitted to JOSES Workshop, January 2001

GeGa01 A.J.C. van Gemund, Hasyim Gautama,*Trade-offs in Symbolic Cost Estimation of Parallel Programs*, submitted to JOSES Workshop, January 2001.

Gemu94 A.J.C. van Gemund, Compiling performance models from parallel programs, in *Proc. 8th ACM International Conf. on Supercomputing* (ICS'94), Manchester, July 1994, ACM, pp. 303-312.

Hagg01 Patrik Hagglund, *Performance Evaluation of Java Implementations using a Base Station*, submitted to JOSES Workshop, January 2001.

Laud01 Peeter Laud, *Analysis for Object Inlining in Java*, submitted to JOSES Workshop, January 2001.

Mart98 Florian Martin. PAG - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer,* **2**(1): 46-67, 1998.

Prob01 Christian Probst, *Flow-Sensitive Call Graph Construction for Java*, submitted to JOSES Workshop, January 2001.

RDKS00 C. van Reeuwijk, W.J.A. Denissen, F. Kuijlman, and H.J. Sips, Annotating Spar/Java for the placement of tasks and data on heterogeneous parallel systems. *In 8th Int. Workshop on Compilers for Parallel Computers CPC'2000*, Aussois, France, January 4-7, 2000.

TiPa00 Frank Tip, Jens Palberg, Scalable propagation-based call graph construction algorithms, in *ACM Sigplan Notices*, 35(10), pp. 281-293, October 2000.

WSR00 R. Wilhelm, M. Sagiv, and T. Reps. Shape Analysis. In *Proceedings of CC 2000: 9th International Conference on Compiler Construction,* Berlin, Germany, March 2000. Spronger-Verlag.

Performance Evaluation of Java Implementations using a Base Station Controller (BSC) Simulator

Patrik Hägglund, `patha@softlab.ericsson.se`

Ericsson SoftLab AB
Linköping, Sweden

2001-01-21

Abstract

A BSC simulator has been benchmarked for the purpose of evaluating Java implementations in the JOSES project. So far five implementations have been evaluated: Sun's JDK 1.2, Sun's Java 2 SDK 1.3 with the Hotspot Client and Server VM, Kaffe 1.0.6, and NewMonics' Perc VM 3.0.1. No major performance difference was shown between these implementations. However, Kaffe showed a small performance lead.

More work remains to be done. The JOSES compiler (when it is available) and some other Java implementations will be included in the evaluation and the benchmark will be improved.

1 Introduction

This work is a part of the JOSES project, and its final purpose is to evaluate the performance of the JOSES compiler for applications written by Ericsson.

The simulator application was written by an application design apartment within Ericsson for the purpose of evaluating Java on TelORB (an Ericsson-specific operating system). The application was also selected before Ericsson SoftLab entered the JOSES project. Therefore, we started the work with the following basic questions:

What to measure? A program that traced all network traffic was constructed. This helped us to understand how the simulator worked. The BSC program keeps a record of the state of all communication channels and handles channel allocation when a mobile phone wants to make a call, and cell handover requests. The time that passes between an incoming request (network message) arrives in the BSC and the corresponding response is sent (over the network) has been recorded (see figure 2).

How to measure? This showed to be a surprisingly hard problem to solve. The application sleeps most of the time, and when there is some CPU activity, it typically lasts for less than one millisecond.

We tried two different approaches: By monitoring the network traffic, and by using a UNIX microsecond-resolution clock, gettimeofday, through

Java Native Interface calls. We also realized that the process that is measured has to be scheduled in a POSIX real-time scheduling class (by using sched_setscheduler) to get repeatable results.

A small client-server application was written to see how much time it takes to just communicate over network sockets in Java.

Why this application? A real BSC is a huge application, and at Ericsson such applications are not currently written in Java. The simulator was chosen because it was assumed that it resembles a real BSC well enough for the purpose of a performance evaluation; possibly after some customization.

However, the BSC simulator does not do much CPU-intensive work and therefore, is not ideal as a Java benchmark. It remains to add some relevant CPU-intensive task of a real BSC.

Class loading has been removed from the BSC simulator to customize it for the JOSES compiler. Unfortunately, the JOSES compiler was not ready when this article was put together, but it will be included in this evaluation as soon as it runs the application.

2 The BSC simulator application

The BSC simulator is accompanied with a MSC (Mobile Switching Center) and one or more BTSs (Base Station Transceiver Systems), that communicate with the BSC. Figure 1 below also shows some switching system components that usually is used in conjunction with an MSC, such as the Visiting Location Register (VLR) and the Home Location Register (HLR), but not included in this simulator.

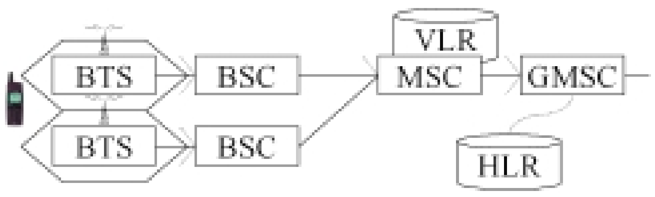


Figure 1: Typical GSM configuration

The Gateway Mobile Switching Center (GMSC) in the figure (which usually is just an ordinary MSC) connects the mobile network to other telephone networks.

The simulator consist of about 10.000 lines of pure Java code. The BSC application itself consist of approximately 5.600 lines of code.

The BTSs connect to the mobile phones (mobile subscribers) and each BTS administrates the radio communication in the cell that belongs to that BTS. A BSC administrates a number of BTSs and collects measurement data that is used to decide when a mobile subscriber is to be moved from one cell (BTS) to another within the area that the BSC administrates.

An MSC administrates a number of BSCs and is more oriented to telephone services than to radio communication. A MSC can also act as a gateway to other networks.

TCP sockets are used for the communication between the programs in this simulator. A TCP connection is used for each of the channels, currently 23 between the BSC and each BTS, and one between the BSC and the MSC. The size of a message varies between 2 to 45 bytes, but is typically about 15 bytes. For each TCP connection the socket option TCP_NODELAY is used to prevent queuing delays of small messages[1].

2.1 Typical behavior of the BSC

The BSC simulator program does not send any messages on its own initiative. It just waits for incoming messages from a BTS or the MSC. When an incoming message arrives it responds with an appropriate answer message as soon as possible.

Figure 2 shows the typical messages that have been implemented in the simulator and which we have measured.

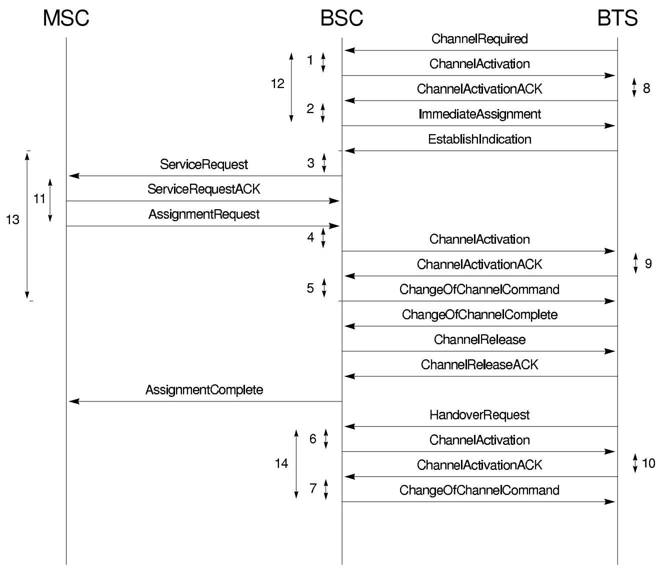


Figure 2: Sequence diagram over measured response times. 1-7 are response times of the BSC. 8-10 are response times of the BTSes. 11 is the response time of the MSC. 12-14 are response times of the BSC for typical message sequences (with external delays included).

The amount of CPU work is low and the response time for any message is typically below one millisecond (on a Pentium III 500MHz running Linux).

The CPU work performed between an incoming message and a response is divided into about 30 to 50 percent for the TCP communication (see section 3)

[1]UDP may have been a more natural protocol for sending small messages, but the specification for the simulator specifies TCP as the protocol to be used.

and the rest for miscellaneous bookkeeping of for example the internal state and the queue of incoming messages.

If the response time from the BSC to a BTS exceeds 200 ms (in some cases 250 ms or 500 ms) the message is resent from the BTS to the BSC. If these time-outs occur too often then BSC is considered to be overloaded. We have tested the BSC with up to 40 BTSs (memory exhaustion prevented us from starting more BTSs) but not seen any sign of the BSC to be overloaded in our measurements.

3 Measurements

The benchmarking was performed on a 500MHz Pentium III running Linux 2.2.18 and glibc 2.2.1. The following Java implementations have been benchmarked:

- Sun's JDK 1.2.2_006 (Interpreted mode, green threads).
- Sun's Java 2 SDK 1.3.0_01 Hotspot Client VM (Adaptive compiler, native threads).
- Sun's Java 2 SDK 1.3.0_01 Hotspot Server VM (Adaptive compiler, native threads).
- Kaffe 1.0.6 (Just-in-time compiler, thread implementation unknown).
- NewMonics' Perc VM 3.0.1 (Just-in-time compiler, green threads).

The Java source code was compiled to byte-code with Sun's JDK 1.2.2_006 for all benchmarks.

To be able to get an idea of how the time is divided into communication time and other kind of CPU work, a small Java server-client application was written and benchmarked too (see section 3.2). This application was also implemented in C to relate the measurements to something that is known to give close to optimal performance.

3.1 The BSC simulator

All processes in the simulator were executed on the same machine. Therefore, the BSC process is executed in a POSIX real-time scheduling class, SCHED_FIFO[2], to make the timing more cohesive and repeatable. (When real-time scheduling on Linux 2.4.0 was used, unexplained network delays that seemed to be multiples of 2 seconds were experienced. Therefore, Linux 2.2.18 was chosen instead.)

10 BTS processes were started in each test and the time samples were collected during approximately one minute. A process that listens to the network traffic is used to collect the time-stamps of each message.

In figure 3 we can see for the response times of the BSC, columns 1-7, that most of the samples are gathered at the lower end, but that some delayed responses deviate from the rest. These delays may be caused by for example

[2]Processes in the SCHED_FIFO scheduling class are assigned a fixed priority and are only preempted by processes with a higher priority (in this case none).

garbage collection in the Java VM or some activity in the Linux kernel. No response comes below 0.2 milliseconds (200 microseconds).

Column 8-11 are response times for the MSC and BTSes. Column 12-14 are "compound" responses from the BSC (see figure 2). Column 15-17 represents the same responses as 12-14 but with delays in external processes (the MSC and BTSes) removed.

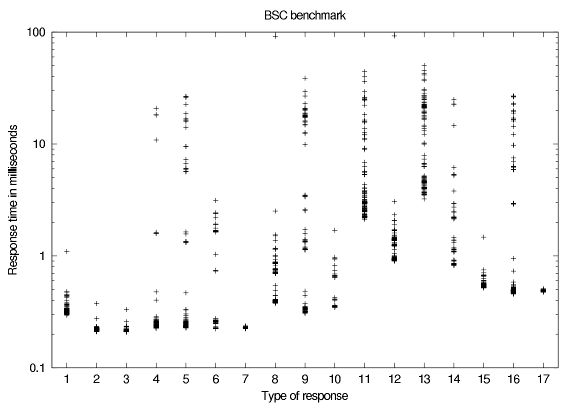


Figure 3: Measured response times for Sun's JDK 1.2.

Unfortunately, the Java SDK 1.3 VM can not be executed under a real-time scheduling class (bug id 114498). This makes the response time more unpredictable and spreads the samples in figure 4 and figure 5.

A second problem with JDK 1.3 was the amount of memory used for each thread. About 70 threads are used for each BTS which caused memory exhaustion when we tried to start 10 BTSes. Therefore, the JDK 1.2 VM was used for the BTSes and the MSC in this test.

No noticeable difference between the client and server version of the Hotspot VM can be seen in this benchmark. The Hotspot Server VM probably need more execution time to show its potential strength.

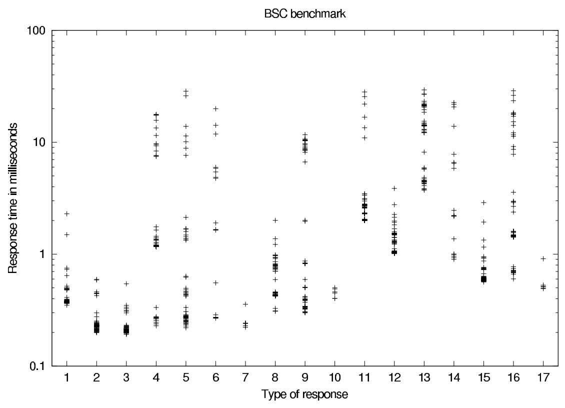


Figure 4: Measured response times for Sun's Java 2 SDK 1.3, Hotspot Client VM.

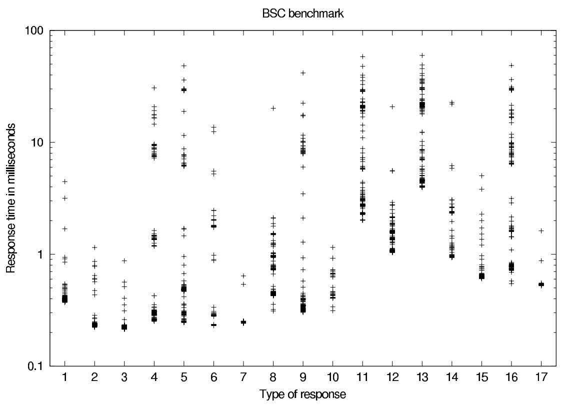


Figure 5: Measured response times for Sun's Java SDK 1.3, Hotspot Server VM.

The Kaffe VM shows a solid behavior and is the Java implementation that gave the best performance in this test.

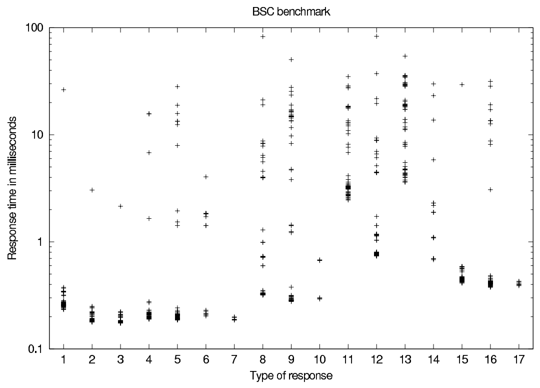


Figure 6: Measured response times for Kaffe 1.0.6.

The Perc VM shows rather bad performance figures in this test, as shown in figure 7. We do not know the reason for this but plan to investigate it.

3.2 The trivial client-server application

The trivial client-server application consists of a server process that listens to a TCP port and echos incoming messages. The client sends 30000 messages of 15 bytes each. The response time of the server is measured by a third process that listens to the network traffic, and the server is scheduled with a real-time scheduling class as described in section 3.1 above.

As for the BSC we can see that responses occasionally are delayed, which probably are caused by Java garbage collection or activity in the Linux kernel. The typical response time is about 100 microseconds, which can be compared to 200 microseconds for the lowest response time in the BSC.

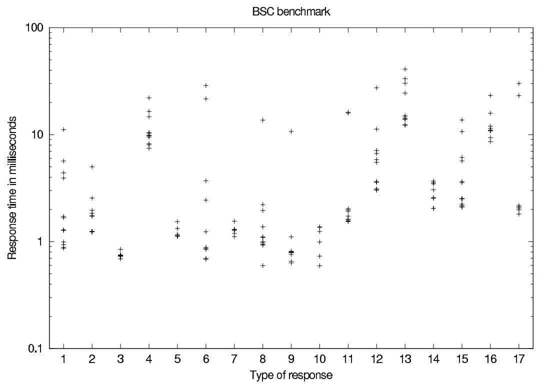


Figure 7: Measured response times for Perc VM 3.0.1.

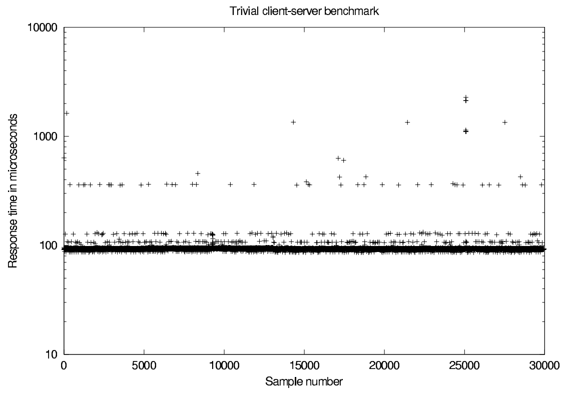


Figure 8: Measured response times for Sun's JDK 1.2

As stated above, the Hotspot VM can not be scheduled in a real-time scheduling class, which makes the samples in figure 9 and 10 quite noisy.

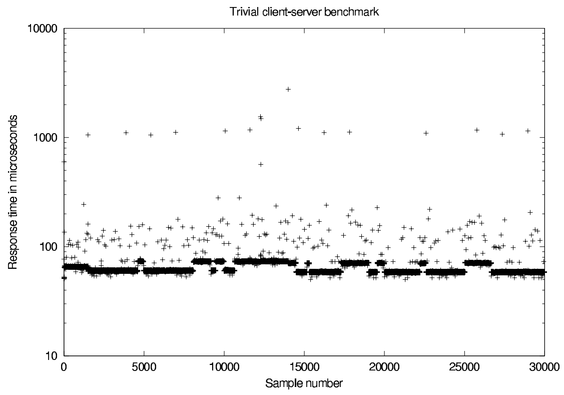


Figure 9: Measured response times for Sun's JDK 1.3, Hotspot Client VM

No significant difference between the Hotspot Client VM and the Hotspot Server VM can be seen in figure 9 and 10.

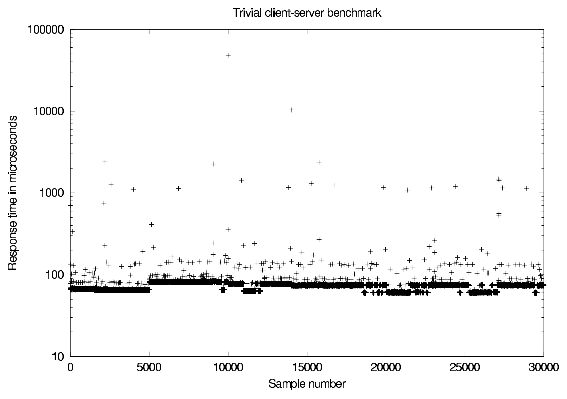


Figure 10: Measured response times for Sun's JDK 1.3, Hotspot Server VM

Kaffe (figure 11) is a few microseconds faster than Sun's JDK 1.2 and the thin line of samples at about 350 microseconds seems to have disappeared.

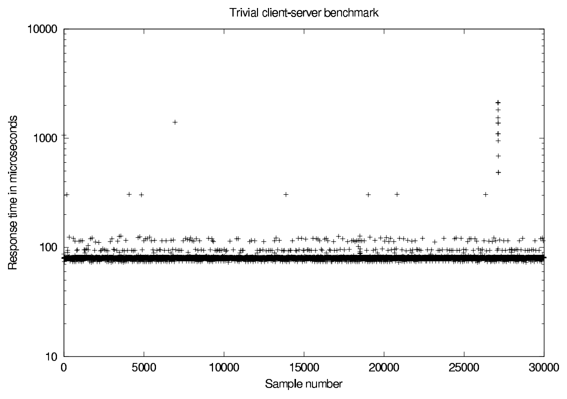


Figure 11: Measured response times for Kaffe

The Perc VM (figure 12) does not seems to get a predictable response time despite the use of a real-time scheduling class. We currently have no explanation for this.

A C implementation of the trivial client-server have also been measured for reference. When this application runs in parallel with a Linux performance monitor, close to 100% of the time is reported to be spent in the operating system kernel.

The typical response time is about 55 microseconds as shown in figure 13. There is no garbage collection in C and all delays therefore have to be due to activity in the Linux kernel.

We can now conclude that the abstraction layers in Java for socket communication adds about 45 microseconds.

The process that listens to the network traffic adds some overhead to the

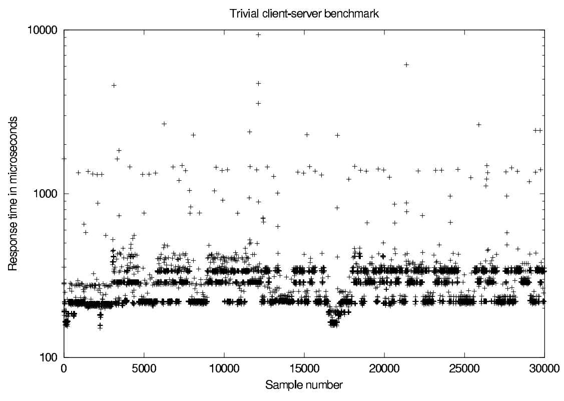


Figure 12: Measured response times for the Perc VM

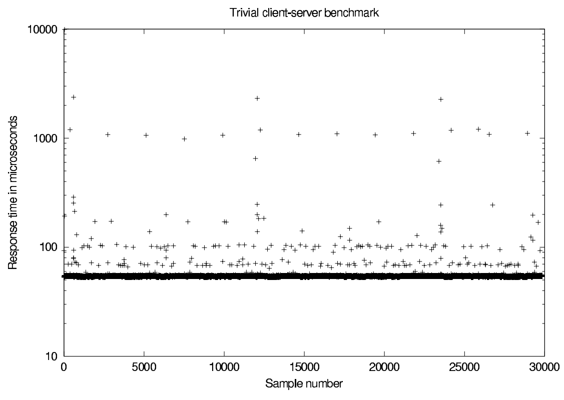


Figure 13: Measured response times for C

network processing time in the kernel. Therefore, a second monitoring approach way evaluated: A microsecond clock (gettimeofday) was called inside the server program just before an incoming message is received and after the response is written. However, the system call that reads from a TCP socket (read) blocks until there is something to read and that idle time can only be excluded by inserting a superfluous second system call (select) that blocks until the data is ready for reading, and starting the clock in between.

This time the measured response time is about 30 microseconds as shown in figure 14, 25 microseconds faster than before.

Unfortunately, this method requires knowledge of the operating system file descriptor associated with the socket used, and this information is not available (without guessing) in Java.

4 Future work

When the JOSES compiler is able to compile any of our programs we will benchmark that compiler. Some more Java implementations will be included

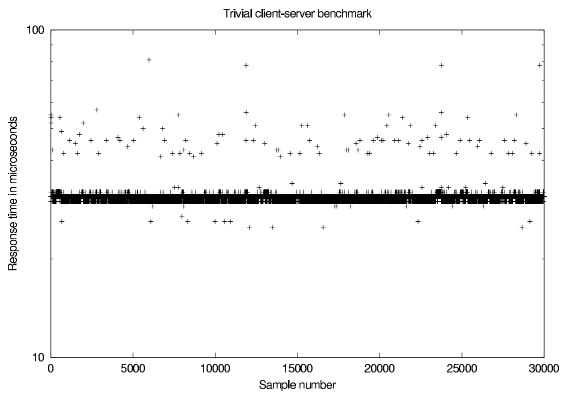


Figure 14: Measured response times for C. Times measured with calls to a microsecond clock within the server program.

too. GCC's Java compiler gcj, and IBM's JDK version 1.1.8 and 1.3 have been benchmarked but the result has not yet included in the report.

We plan to use some standard benchmark suite to get a more complete picture of the differences between the implementations.

We will also investigate how the BSC application can be extended to include more CPU intensive operations.

5 Conclusions

No major performance difference was shown between the implementations that have been evaluated. However, Kaffe showed a small performance lead.

For the simple client-server application, a noticeable performance gap was shown for Java compared to C.

Acknowledgments

I would like to thank my manager Pär Emanuelson for helping me to get some structure to this work.

Some Measurements of Java-to-bytecode Compiler Performance in the Java Virtual Machine

Charles Daly Computer Applications, Dublin City University, Dublin 9, Ireland.
Jane Horgan Computer Applications, Dublin City University, Dublin 9, Ireland.
James Power Dept of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.
John Waldron Department of Computer Science, Trinity College, Dublin 2, Ireland.

ABSTRACT

In this paper we present a platform independent analysis of the dynamic profiles of Java programs when executing on the Java Virtual Machine. The Java programs selected are taken from the Java Grande Forum benchmark suite, and five different Java-to-bytecode compilers are analysed. The results presented describe the dynamic instruction usage frequencies.

These results, presenting a picture of the actual (rather than presumed) behaviour of the JVM, have implications both for the coverage aspects of the Java Grande benchmark suites, for the performance of the Java-to-bytecode compilers, and for the design of the JVM.

KEYWORDS

Java Virtual Machine Interpreter

1 INTRODUCTION

The Java paradigm for executing programs is a two stage process. Firstly the source is converted into a platform independent intermediate representation, consisting of bytecode and other information stored in class files. The second stage of the process involves hardware specific conversions, perhaps by a JIT compiler for the particular hardware in question, followed by the execution of the code. The problem addressed by this research is that while there exist static tools such as class file viewers to look at this intermediate representation, there is currently no easy way of studying the dynamic behaviour at this point in the program. This research therefore sets out to perform dynamic analysis at the platform independent level and investigate whether or not useful results can be gained. In order to test the technique, the Java Grande Forum's Benchmark suite was used.

The remainder of this paper is organised as follows. Section 2 discusses the background to this work, including the rationale behind bytecode-level dynamic analysis, and the test suite used. Sections 3 and 4 summarise the profiles of each of the Grande programs studied. In particular, section 3 presents a method-level view of the dynamic profile, while section 4 presents a more detailed bytecode-level view. Section 5 discusses the influence of compiler choice on dynamic analysis, and describes the variances caused by five of the most common Java compilers. Section 7 concludes the paper.

E-mail address for correspondence:
John.Waldron@cs.tcd.ie

2 BACKGROUND

The increasing prominence of internet technology, and the widespread use of the Java programming language has given the Java Virtual Machine (JVM) a unique position in the study of compilers and related technologies. To date, much of this research has concentrated on the performance of the bytecode interpreter, yielding techniques such as Just-In-Time (JIT) and hotspot-centered compilation.

However, the production of bytecode for the JVM is no longer limited to a single Java-to-bytecode compiler. Not only is there a variety of different Java compilers available, but there are also compilers for extensions and variations of the Java programming language, as well as for other languages such as Eiffel and Scheme, all targeted on the JVM. In previous work we have studied the impact of the choice source language on the dynamic profiles of programs running on the JVM [2]. In this paper we examine the impact of the choice of Java compiler on the dynamic execution of JVM bytecodes, and analyse the degree to which the Java Grande [1] applications can fulfill the role as a standard test suite for these and other aspects of the JVM.

2.1 Dynamic Bytecode-Level Analysis

The output of a dynamic bytecode analysis will therefore be important for the design of both Java to bytecode and Just-In-Time bytecode to native compilers. Of particular interest also is the instruction set used by an intermediate representation to implement platform independence. By dynami-

mol methods	freq
java/lang/Math.sqrt†	19.4
moldyn/particle.velavg	18.8
moldyn/particle.mkekin	18.8
moldyn/particle.force	18.8
moldyn/particle.domove	18.8
moldyn/random.update	1.4
moldyn/random.seed	0.6
java/lang/Math.log†	0.6

eul methods	frequency
java/lang/Math.abs	24.5
java/lang/Object.<init>	19.6
euler/Statevector.<init>	19.5
euler/Statevector.svect	19.2
java/lang/Math.sqrt†	11.5
euler/Vector2.dot	1.8
euler/Vector2.magnitude	1.4
java/lang/Math.pow†	0.3

sea methods	freq
search/Game.wins	46.5
search/SearchGame.ab	10.3
search/Game.makemove	10.3
search/Game.backmove	10.3
search/TransGame.hash	9.3
search/TransGame.transpose	5.3
search/TransGame.transtore	4.0
search/TransGame.transput	4.0

ray methods	frequency
raytracer/Vec.dot	47.0
raytracer/Vec.sub2	23.2
raytracer/Sphere.intersect	22.8
java/lang/Math.sqrt†	1.6
java/lang/Object.<init>	1.3
raytracer/Vec.<init>	0.7
raytracer/Vec.normalize	0.6
raytracer/Isect.<init>	0.6

Table 1: *Dynamic method execution frequencies for the most heavily used methods for the Grande application including native methods, indicated by* †.

cally analysing the Java bytecodes, lessons may be drawn to facilitate construction of more efficient intermediate representations for both procedural object-oriented programming languages like Java and programming languages from different categories.

Speed comparisons of the Java Grande benchmark suite using different Java Platforms have been performed [1] and differences in execution times have been found, but it has not been known whether the resulting differences measured have been due to the Java compiler, the JIT compiler or the virtual machine implementation on the particular underlying operating system and hardware architecture. This paper shows, by means of the dynamic bytecode analysis technique, that the bytecodes executed by a particular Grande application are very similar for a wide variety of Java compilers, implying compiler choice is not the main explanation of execution speed variations for these programs. In addition, it is possible to study how representative of Grande programs those chosen for the benchmark suite are.

In order to study dynamic bytecode usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [3] is an independent implementation of the Java Virtual Machine which was written from scratch and is free from all third party royalties and license restrictions. It comes with its own standard class libraries, including Beans and Abstract Window Toolkit (AWT), native libraries, and a highly configurable virtual machine with a JIT compiler for enhanced performance. Kaffe is available under the Open Source Initiative and comes with complete source code, distributed under the GNU Public License. Version: 1.0.5 was used for these measurements.

2.2 Grande Programs Measured

A *Grande* application is one which uses large amounts of processing, I/O, network bandwidth or memory. The Java Grande Forum Benchmark Suite (http://www.epcc.ed.ac.uk/javagrande/) is intended to be representative of such applications, and thus to provide a basis for measuring and comparing alternative Java execution environments. It is intended that the suite should include not only applications in science and engineering but also, for example, corporate databases and financial simulations.

- The **moldyn** benchmark is a translation of a Fortran program designed to model the interaction of molecular particles. Its origin as non object-oriented code probably explains its relatively unusual profile, with few methods which make intensive use of fields within the class, even for temporary and loop-control variables. This program may still represent a large number of Grande type applications that will initially run on the JVM

- The **search** benchmark solves a game of connect-4 on a 6×7 board using alpha-beta pruning. Intended to be memory and numerically intensive, this is also the only application to demonstrate an inheritance hierarchy of depth greater than 2.

- The **euler** benchmark solves a set of equations using a fourth order Runge-Kutta method. This suite demonstrates a considerable clustering of functionality in the `Tunnel` class, as well as a comparatively high percentage of methods with very large local variable requirements.

- The **raytracer** measures the performance of a 3D ray tracer rendering a scene containing 64 spheres. It is represented using a fairly shallow inheritance tree, with functionality (as measured in methods) fairly well distributed throughout the classes.

Program	Total methods	API %	API native %
mol	5.45e+05	22.0	20.0
sea	7.12e+07	0.0	0.0
eul	3.34e+07	58.0	12.6
ray	4.58e+08	3.1	1.6
average	1.41e+08	20.8	8.6

Table 2: *Measurements of total number of method calls including native calls by Grande applications.* Also shown is the percentage of the total which are in the API, and percentage of total which are in API and are native methods.

Program	Java method calls		bytecodes executed	
	number	% in API	number	% in API
mol	4.36e+05	2.5	7.60e+09	0.0
sea	7.12e+07	0.0	7.39e+09	0.0
eul	2.92e+07	51.9	1.58e+10	20.9
ray	4.50e+08	1.5	1.18e+10	0.8
average	1.38e+08	14.0	1.06e+10	5.4

Table 3: *Measurements of Java method calls excluding native calls made by Grande applications.*

Program	io	lang	net	text	util
mol	4.0	74.7	1.1	0.4	19.8
sea	4.9	68.4	1.5	0.0	25.2
eul	2.4	97.5	0.0	0.0	0.0
ray	0.0	100.0	0.0	0.0	0.0

Table 4: *Breakdown of Java API method dynamic usage percentages by package for Grande applications.* None of the applications used methods from the applet, awt, beans, math, security or sql packages.

- The **montecarlo** benchmark is a financial simulation using Monte Carlo techniques to price products derived from the price of an underlying asset. Its use of classical object-oriented get and set methods accounts for the relatively high proportion of methods with no temporary variables and 1 or 2 parameters (including the `this`-reference).

3 DYNAMIC METHOD EXECUTION FREQUENCIES

In this section we present our first dynamic profile of the Grande programs studied. Here we partition the execution profiles based on methods, since these provide both a logical level of modularity at source-code level, as well as a likely unit of granularity for hotspot analysis. It should be noted that these figures are not the usual *time-based* analysis, which will vary considerably between different computer configurations and architectures, but are based on the more platform-independent *bytecode frequency* analysis.

As Kaffe and the JVM are not yet mature technologies for Grande applications, some programs in the suite fail to compute the correct result. It was decided to exclude the *montecarlo* benchmark from this study as it failed by a large amount when interpreted, but *raytracer* was included as the error in the result was very small.

Table 1 shows dynamic method execution frequencies for the most heavily used methods for the Grande applications including native methods. It can be seen that virtually all execution time is spent in at most five methods for these applications.

Table 2 shows measurements of the total number of method calls including native calls by Grande applications. For the programs studied, on average 8.6% of methods are API methods which are implemented by native code. As the benchmark suite is written in Java it is possible to conclude that any native methods are in the API. This paper is confined to studying how the Java methods execute.

Table 3 shows measurements of the Java method calls excluding native calls. With the exception of the *eul* benchmark, Java method execution time is virtually all in the non-API bytecodes of the programs. This is a significant difference from traditional Java applications such as applets or compiler type tools which spend most of the time in the API [4]. Mixed compiled interpreted systems which precompile the API methods to some native format will therefore not be as effective at speeding up Grande applications like these. The finding that API usage is very low may imply that the benchmark suite may not be fully representative of a broad range of Grande applications (see Table 4). It is also possible to observe that since 51% of Java methods are API for the *eul* benchmark, but only 21% of the bytecodes executed, that the API methods are smaller in size than the Grande program's methods. All measurements in this paper were made with the Kaffe API library, which may differ from other Java API libraries.

Table 4 shows dynamic measurements of the Java API package method percentages. As would be expected for the programs considered, the applet and awt packages are not used at all as graphics has been removed from the benchmarks. Of major interest is that the math package is not used by the benchmarks which implies either the benchmarks are not representative of numerical programs or the math package is not in fact of much use to such programs which simply use the java.lang.Math class. A Grande application should use large amounts of processing, I/O, network bandwidth or memory, yet it is interesting to note how little of the API packages are dynamically used by this benchmark suite.

4 DYNAMIC BYTECODE EXECUTION FREQUENCIES

In this section we present a more detailed view of the dynamic profiles of the Grande programs studied by considering the frequencies of the different bytecodes used. These figures help to provide a detailed description of the nature of the operations being performed by each program, and thus give a picture of the aspects of the JVM actually being tested by the suite. This also provides an alternative to typical time-based analysis, which, while useful for efficiency analysis, can be considerably influenced by the underlying architecture's proficiency in dealing with different types of

mol		sea		eul		ray	
dload	33.3	iload	13.2	iload	19.7	getfield	26.1
iload	7.0	aload_0	8.6	aaload	18.2	aload_0	16.1
dstore	6.8	getfield	7.3	getfield	16.2	aload_1	10.9
dcmpg	5.5	iaload	5.4	aload_0	8.3	dmul	6.5
dsub	4.7	istore	5.3	dmul	4.1	dadd	4.7
dmul	4.3	ishl	4.3	dadd	4.0	dsub	3.7
getstatic	4.3	bipush	3.8	putfield	3.3	putfield	3.1
getfield	4.3	iload_1	3.6	iconst_1	3.2	aload_2	2.8
aaload	4.2	iand	3.5	dload	2.8	dreturn	1.9
dneg	4.1	iadd	3.5	isub	2.0	invokevirtual	1.9
dcmpl	4.1	iload_2	2.6	daload	2.0	invokestatic	1.9
ifge	4.1	iload_3	2.5	dup	1.7	dload_2	1.9
ifle	4.1	ior	2.3	aload_3	1.5	iload	1.8
dadd	3.4	iconst_1	2.3	dsub	1.4	aload	1.3
iinc	1.4	iconst_2	2.1	aload	1.3	dload	1.1
ifgt	1.4	dup	2.0	aload_2	1.3	dconst_0	1.0
if_icmplt	1.4	iinc	1.7	ldc2w	1.1	dcmpg	1.0
dload_1	1.0	ifeq	1.6	iload_3	1.1	ifge	1.0
putfield	0.1	iastore	1.5	iadd	1.1	return	1.0
aload_0	0.1	if_icmplt	1.4	dstore	1.0	dstore	1.0
nop	0.0	iconst_4	1.4	ddiv	0.6	iinc	0.9
isub	0.0	iconst_5	1.4	dconst_0	0.4	if_icmplt	0.9
lsub	0.0	if_icmple	1.3	aload_1	0.4	areturn	0.9
fsub	0.0	invokevirtual	1.0	iinc	0.3	arraylength	0.9
imul	0.0	dup2	1.0	if_icmplt	0.3	ifnull	0.9
lmul	0.0	isub	0.9	dload_1	0.3	aconst_null	0.9
fmul	0.0	if_icmpgt	0.9	dload_3	0.3	aaload	0.9
idiv	0.0	ldc1	0.8	dstore_1	0.2	astore	0.9
ldiv	0.0	istore_3	0.8	dstore_3	0.2	dstore_2	0.9
lconst_1	0.0	imul	0.7	dastore	0.2	dload_1	0.2
fdiv	0.0	ifne	0.7	dneg	0.1	ddiv	0.1
ddiv	0.0	putfield	0.7	dcmpg	0.1	dcmpl	0.1
irem	0.0	iconst_0	0.7	ifge	0.1	ifle	0.1
lrem	0.0	istore_1	0.7	if_icmpge	0.1	goto	0.1
frem	0.0	if_icmpne	0.6	if_icmple	0.1	invokespecial	0.1

Table 5: *Total (API and non-API) dynamic bytecode usage frequencies by Grande applications compiled using SUN's javac compiler, Standard Edition (JDK build 1.3.0-C)* The top 35 instructions are presented.

bytecode instructions.

Table 5 shows total (API and non-API) dynamic bytecode usage frequencies by Grande applications. The JVM instruction set has special efficient load and store instructions for the first four local variable array entries, and less efficient generic instructions for higher local variable array positions. The first thing that stands out from Table 5 is that for *mol, sea* and *eul* the highest frequency instruction is a generic load, rather than an efficient load from one of the first four elements of the local variable array. For *mol* one third of instructions are a single load of this type.

Although the Java to bytecode compiler does not have access to dynamic execution data, it should be able to put the most heavily used local variable into one of the efficient slots most of the time (see also Table 8). Alternatively, if the compiler just assigns the local variables in the order they are declared, the application programmer might be able to alter the sequence to increase efficiency in some cases, but not if the compiler always puts the parameters first and there are a large number of these.

The *mol* benchmark has the same number of `getfield` as `getstatic` instructions, uses a much smaller set of instruction than the other benchmarks, and does not have method invocations in its high frequency instructions, suggesting it may not have been designed in an object-oriented fashion. The comparison instruction `dcmpl` is also at very high frequency in *mol* relative to the other benchmarks, suggesting something different is happening in the structure of the code involving a high number of dynamic decisions. `invokevirtual` does not appear at all in the high frequency instruction for *eul* or *mol*, and is at 1% for *sea* and 1.9% for *ray* suggesting that worries about the inefficiencies of virtual method invocation in the Java language may have been overstated for Grande applications. Of course, the execution time for the `invokevirtual` instruction will be much higher than for ordinary instructions on any hardware platform. *ray* seems to be the most object-oriented program, using `getfield` as its most frequent instruction, followed by `aload_0` to access the `this`-reference.

In order to study overall bytecode usages across the pro-

Compiler	mol	eul	sea	ray
kopi	7599606497	12475753926	7388409738	11706547525
pizza	7704747144	11431095142	7311241755	11919084828
gcj	7704740202	12540807644	7527673585	11810849733
jdk13	7599606435	11394409844	7103719939	11706547247
borland	7705054344	11431120742	7324210788	11919084856

Table 6: *Total Non-API dynamic bytecode usage counts for Grande Applications using different compilers. For* gcj*, a minor alteration to the* sea *program source was needed to get it to compile.*

grams, it is possible to calculate the average bytecode frequency

$$f_i = \frac{1}{n}\sum_{k=1}^{n} \frac{100 \times c_{ik}}{\sum_{i=1}^{256} c_{ik}}$$

where c_{ik} is the number of times bytecode i is executed during the execution of program k and n is the number of programs averaged over. f_i is an approximation of that bytecode's usage for a typical Grande program.

5 COMPARISONS OF DYNAMIC BYTECODE USAGES ACROSS DIFFERENT COMPILERS

In this section we consider the impact of the choice of Java compiler on the dynamic bytecode frequency figures. Java is relatively unusual (as compared to, say, C or C++) in that optimisations can be implemented in two separate phases: first when the source program is compiled into bytecode, and again when this bytecode is executed on a specific JVM. We consider here those optimisation which are implemented at the compiler level, and thus may be considered to be platform independent, and which must be taken into account in any study of the bytecode frequencies.

For the purposes of this study we used five different Java compilers, from the following development environments:

kopi KOPI Java Compiler Version 1.3C
`http://www.dms.at/kopi`

pizza Pizza version 0.39g, 15-August-98
`http://www.cis.unisa.edu.au/~pizza/`

gcj The GNU Compiler for the Java Programming Language version 2.95.2
`http://sources.redhat.com/java/`

jdk13 SUN's javac compiler, Standard Edition (JDK build 1.3.0-C)

borl Borland Compiler 1.2.006 for Java

The figures for the Java compiler from 1.2 of SUN's JDK, as well as version 1.06 of the IBM Jikes Compiler were also computed, but since the code produced was almost identical to that produced by the compiler from version 1.3 of the JDK we do not consider them further here.

Table 6 shows total Non-API dynamic bytecode counts for the Grande programs using different compilers. The API was not recompiled and those bytecodes were excluded from the dynamic comparisons. While it is difficult to draw direct conclusions based on these figures, two facts are at least apparent. First, examining each column of Table 6, it can be seen that there are significant differences between the bytecodes executed for a single application between the different compilers. Second, this variance is not consistent through all four applications, and it is clear that a more detailed analysis is necessary to account for these differences.

Ideally, the optimisations implemented by each compiler should be described in the corresponding documentation; regrettably this is not the case in reality. Also, since each of the applications produces significantly large bytecode files, a static analysis of the differences between these files is not practical. Further, a bytecode-level static analysis would not be sufficient for determining those differences which resulted in a significant variance in the dynamic profiles.

Instead, a detailed analysis of the dynamic bytecode executed frequencies was carried out. The raw statistics are presented in Table 7, Table 8, Table 9 and Table 10, which show the top 35 most executed instructions for each application. In order to analyse these tables, the differences in each row were selected, and the relevant sections of the corresponding source code was then examined. Below we summarise the main differences exhibited in these tables.

5.1 Main Compiler Differences

There were three main differences between the optimisations implemented by the compilers:

Loop Structure The figures show a difference in the use of comparison and jump instructions between the compilers. For each usage of the `if_cmplt` instruction by *kopi* and *jdk13* there is a corresponding usage of `goto` and `if_cmpge` by *pizza*, *gcj* and *borland*. This can be explained by the implementation of loop structures. for example, a loop of the form:

`while (expr) { stats }`

is implemented by the different compilers as follows:

kopi/jdk13	*pizza/gcj/borland*

`      goto end`
`beg:  stats`
`end:  expr`
`      if_cmplt beg`
	`beg:  expr`
	`      if_cmpge end`
	`      stats`
	`      goto beg`
	`end:`

A simple static analysis would regard these as similar implementations, but the dynamic analysis clearly shows the savings resulting from the *kopi/jdk13* approach.

Specialised `load` Instructions Table 8 and Table 9 highlight an important difference between the compilers in their

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
dload	33.3	32.8	32.8	33.3	32.8	33.0
iload	7.0	6.9	6.9	7.0	6.9	6.9
dstore	6.8	6.7	6.7	6.8	6.7	6.7
dcmpl	9.7	4.1	4.1	4.1	4.1	5.2
dsub	4.7	4.7	4.7	4.7	4.7	4.7
dmul	4.3	4.3	4.3	4.3	4.3	4.3
dcmpg	0.0	5.4	5.4	5.5	5.4	4.3
getstatic	4.3	4.2	4.2	4.3	4.2	4.2
getfield	4.3	4.2	4.2	4.3	4.2	4.2
aaload	4.2	4.2	4.2	4.2	4.2	4.2
dneg	4.1	4.1	4.1	4.1	4.1	4.1
ifge	4.1	4.1	4.1	4.1	4.1	4.1
ifle	4.1	4.1	4.1	4.1	4.1	4.1
dadd	3.4	3.4	3.4	3.4	3.4	3.4
iinc	1.4	1.4	1.4	1.4	1.4	1.4
ifgt	1.4	1.4	1.4	1.4	1.4	1.4
dload_1	1.0	1.0	1.0	1.0	1.0	1.0
if_icmpge	0.0	1.4	1.4	0.0	1.4	0.8
goto	0.0	1.4	1.4	0.0	1.4	0.8
if_icmplt	1.4	0.0	0.0	1.4	0.0	0.6
putfield	0.1	0.1	0.1	0.1	0.1	0.1
aload_0	0.1	0.1	0.1	0.1	0.1	0.1
nop	0.0	0.0	0.0	0.0	0.0	0.0
isub	0.0	0.0	0.0	0.0	0.0	0.0
lsub	0.0	0.0	0.0	0.0	0.0	0.0
fsub	0.0	0.0	0.0	0.0	0.0	0.0
imul	0.0	0.0	0.0	0.0	0.0	0.0
lmul	0.0	0.0	0.0	0.0	0.0	0.0
fmul	0.0	0.0	0.0	0.0	0.0	0.0
idiv	0.0	0.0	0.0	0.0	0.0	0.0
ldiv	0.0	0.0	0.0	0.0	0.0	0.0
lconst_1	0.0	0.0	0.0	0.0	0.0	0.0
fdiv	0.0	0.0	0.0	0.0	0.0	0.0
ddiv	0.0	0.0	0.0	0.0	0.0	0.0
irem	0.0	0.0	0.0	0.0	0.0	0.0

Table 7: *Non-API dynamic bytecode usage frequencies for* **mol** *using different compilers.* The top 35 instructions are presented.

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
aaload	21.6	19.8	21.5	19.9	19.8	20.5
iload	22.4	20.7	5.4	20.8	20.7	18.0
getfield	17.3	17.0	17.4	17.0	17.0	17.1
aload_0	10.0	9.0	10.1	9.0	9.0	9.4
dadd	3.8	4.1	3.8	4.1	4.1	4.0
dmul	3.7	4.1	3.7	4.1	4.1	3.9
iconst_1	2.7	2.9	2.7	2.9	2.9	2.8
putfield	2.6	2.8	2.6	2.8	2.8	2.7
dload	2.5	2.7	2.9	2.7	2.7	2.7
iload_3	1.3	1.4	7.7	1.4	1.4	2.6
isub	1.7	1.9	1.7	1.9	1.9	1.8
iload_2	0.0	0.0	9.1	0.0	0.0	1.8
aload_3	1.7	1.9	1.7	1.9	1.9	1.8
daload	1.6	1.8	1.6	1.8	1.8	1.7
dup	0.1	2.0	0.1	2.0	2.0	1.2
dstore	1.0	1.1	1.4	1.1	1.1	1.1
dsub	0.9	1.0	0.9	1.0	1.0	1.0
ldc2w	1.0	1.1	0.7	1.1	1.1	1.0
iadd	1.0	1.0	0.9	1.0	1.0	1.0
ddiv	0.6	0.7	0.6	0.7	0.7	0.7
aload_2	0.3	0.4	0.3	0.4	0.4	0.4
iinc	0.3	0.3	0.3	0.3	0.3	0.3
iload_1	0.0	0.0	1.4	0.0	0.0	0.3
dconst_0	0.2	0.2	0.2	0.2	0.2	0.2
if_icmpge	0.0	0.3	0.3	0.0	0.3	0.2
goto	0.0	0.3	0.3	0.0	0.3	0.2
dload_1	0.2	0.3	0.0	0.3	0.3	0.2
dload_3	0.2	0.2	0.0	0.2	0.2	0.2
aload_1	0.2	0.2	0.2	0.2	0.2	0.2
dstore_1	0.2	0.2	0.0	0.2	0.2	0.2
dstore_3	0.2	0.2	0.0	0.2	0.2	0.2
dastore	0.2	0.2	0.2	0.2	0.2	0.2
if_icmplt	0.3	0.0	0.0	0.3	0.0	0.1
invokespecial	0.1	0.1	0.1	0.1	0.1	0.1
new	0.1	0.1	0.1	0.1	0.1	0.1

Table 8: *Non-API dynamic bytecode usage frequencies for* **eul** *using different compilers.* The top 35 instructions are presented.

treatment of specialised `iload` instructions. *gcj* gives a significantly lower usage of the generic `iload` instruction relative to all other compilers, and a corresponding increase in the more specific `iload_2` and `iload_3` instructions showing that this compiler is attempting to optimise the programs for integer usage.

However, it is interesting to note the failure of this approach as demonstrated by Table 7 and Table 10, where the differences in `iload` instructions are not significant. This can be explained directly by the nature of the programs involved - *mol* and *ray* make greater use of `doubles` and objects respectively, and *gcj* makes no attempt to optimise the stack positions for these types.

Usage of the `dup` Instruction There is a dramatic difference in the use of `dup` instructions show in Table 8 and, to a lesser extent, in Table 9, with *kopi* and *gcj* having a much lower usage than the other compilers. (`dup` instructions do not account for a significant proportion of bytecode usage in the other applications). This can be explained by the usage of the shorthand arithmetic instructions (such as `+=`) in the source Java code. For example, the *eul* suite contains lines of the form:

```
r[i][j].a += ...
```

A simple translation of this line to the longer form

```
r[i][j].a = r[i][j].a + ...
```

results in code which references the expression `r[i][j].a` twice.

The *pizza*, *jdk13* and *borland* compilers optimise for the first form by duplicating the value of the expressions. The other two compilers do not, and show a corresponding increase in the usages of `aload`, `aaload` and `getfield` instructions.

The presence of the line in what is evidently a program hotspot gives particular relevance to this compiler optimisation in this case.

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
iload	13.4	12.9	12.4	13.2	12.8	12.9
aload_0	9.6	8.3	8.9	8.6	8.3	8.7
getfield	7.9	7.1	7.6	7.3	7.1	7.4
iaload	5.2	5.2	5.1	5.4	5.2	5.2
istore	5.1	5.2	5.2	5.4	5.2	5.2
ishl	4.1	4.2	4.1	4.3	4.2	4.2
bipush	3.6	3.7	4.3	3.8	3.6	3.8
iadd	4.2	3.4	4.1	3.5	3.4	3.7
iand	3.3	3.4	4.1	3.5	3.4	3.5
iload_1	3.8	3.5	2.8	3.6	3.5	3.4
iload_2	2.5	2.6	3.3	2.6	2.5	2.7
iload_3	2.7	2.5	3.3	2.5	2.5	2.7
ior	2.2	2.3	2.2	2.3	2.3	2.3
iconst_1	2.0	2.2	2.0	2.3	2.2	2.1
iconst_2	2.0	2.0	2.0	2.1	2.0	2.0
dup	1.5	1.9	1.8	2.0	1.9	1.8
iinc	1.7	1.7	1.6	1.7	1.7	1.7
iconst_5	1.8	1.4	1.7	1.4	1.4	1.5
iconst_0	0.7	2.5	0.7	0.7	2.6	1.4
iastore	1.4	1.4	1.4	1.5	1.4	1.4
iconst_4	1.4	1.4	1.4	1.4	1.4	1.4
if_icmpgt	0.9	1.7	1.4	0.9	1.7	1.3
goto	0.8	1.5	1.5	0.5	1.5	1.2
ifeq	1.2	0.1	1.9	1.6	0.1	1.0
invokevirtual	1.0	1.0	0.9	1.0	1.0	1.0
isub	0.9	0.9	0.8	0.9	0.9	0.9
if_icmple	1.3	0.6	0.8	1.3	0.6	0.9
if_icmpeq	0.2	1.7	0.2	0.2	1.7	0.8
if_icmplt	1.3	0.5	0.5	1.4	0.5	0.8
ldc1	0.8	0.8	0.8	0.8	0.9	0.8
istore_3	0.8	0.8	0.8	0.8	0.8	0.8
imul	0.6	0.7	0.6	0.7	0.7	0.7
if_icmpge	0.1	1.1	0.9	0.1	1.1	0.7
putfield	0.7	0.7	0.7	0.7	0.7	0.7
dup2	0.1	1.0	0.3	1.0	1.0	0.7

Table 9: *Non-API dynamic bytecode usage frequencies for* **sea** *using different compilers.* The top 35 instructions are presented. For *gcj*, a minor alteration to the program source was needed to get it to compile.

Instruction	kopi	pizza	gcj	jdk13	borl	f_i
getfield	26.3	25.8	26.0	26.3	25.8	26.0
aload_0	16.2	15.8	16.0	16.1	15.8	16.0
aload_1	10.9	10.7	10.8	10.9	10.7	10.8
dmul	6.6	6.5	6.5	6.6	6.5	6.5
dadd	4.7	4.6	4.7	4.7	4.6	4.7
dsub	3.7	3.6	3.7	3.7	3.6	3.7
putfield	3.0	3.0	3.0	3.0	3.0	3.0
aload_2	2.8	2.7	2.8	2.8	2.7	2.8
invokestatic	1.9	1.9	1.9	1.9	1.9	1.9
dreturn	1.9	1.8	1.8	1.9	1.8	1.8
invokevirtual	1.9	1.8	1.8	1.9	1.8	1.8
iload	1.9	1.8	1.8	1.9	1.8	1.8
dload	1.1	1.1	2.9	1.1	1.1	1.5
dload_2	1.9	1.8	0.0	1.9	1.8	1.5
aconst_null	0.9	1.7	0.9	0.9	1.7	1.2
aload	1.2	1.2	1.2	1.2	1.2	1.2
dstore	1.0	1.0	1.8	1.0	1.0	1.2
ifge	1.0	1.0	1.0	1.0	1.0	1.0
iinc	0.9	0.9	0.9	0.9	0.9	0.9
dconst_0	0.9	0.9	0.9	0.9	0.9	0.9
areturn	0.9	0.9	0.9	0.9	0.9	0.9
return	0.9	0.9	0.9	0.9	0.9	0.9
arraylength	0.9	0.9	0.9	0.9	0.9	0.9
aaload	0.9	0.9	0.9	0.9	0.9	0.9
astore	0.9	0.9	0.9	0.9	0.9	0.9
dcmpg	0.0	1.0	1.0	1.0	1.0	0.8
dstore_2	0.9	0.9	0.0	0.9	0.9	0.7
goto	0.1	0.9	1.0	0.1	0.9	0.6
if_icmpge	0.0	0.9	0.9	0.0	0.9	0.5
ifnull	0.9	0.0	0.9	0.9	0.0	0.5
if_icmplt	0.9	0.0	0.0	0.9	0.0	0.4
if_acmpeq	0.0	0.9	0.0	0.0	0.9	0.4
dcmpl	1.1	0.1	0.1	0.1	0.1	0.3
dload_1	0.2	0.2	0.2	0.2	0.2	0.2
ddiv	0.1	0.1	0.1	0.1	0.1	0.1

Table 10: *Non-API dynamic bytecode usage frequencies for* **ray** *using different compilers.* The top 35 instructions are presented.

5.2 Minor compiler differences

Some minor differences between the frequencies can also be noted as follows:

Comparisons with 0 and `null` As well as generic comparison instructions for each type, Java bytecode has two specialised instructions for comparison with zero: `ifeq` and `ifne`. As can be seen from Table 9, the frequencies for these instructions for both the *pizza* and *borland* compilers is lower than the other compilers, and a price is paid in a correspondingly higher use of `iconst_0` and `if_icmpeq` instructions.

As before, this variance is shown to differing degrees dependent on the application: none of the other three programs rate this difference as significant. However, Java bytecode also has a specialised instruction for comparing object references with null, `ifnull`. The object-intensive program *ray* (Table 10) exhibits the results of the *pizza* and *borland* compilers not using this instruction, with a corresponding increase in `aconst_null` and `if_acmpeq` instructions.

The Decrement Instruction There are two approaches to decrementing an integer value. Either you can push minus 1 and add (`iconst_m1`, `iadd`), or push 1 and subtract (`iconst_1`, `isub`). Only the *kopi* and *gcj* compilers choose the former, and so Table 9 shows an increase in the use of `iadd` instructions, along with a corresponding drop in the use of `iconst_1` instructions.

Constant Propagation The *gcj* compiler does not do as much constant propagation as the other compilers and this is evidenced in Table 8. The *eul* application has a number of constant fields, and this is reflected by a drop in `ldc2w` instructions, and a corresponding increase in the number of `getfield` instructions.

Comparison operations A minor variation is shown in Table 7 for the usages of `dcmpl` and `dcmpg` instructions, with the *kopi* compiler showing a strong preference for the former; the dependent statement blocks in the corresponding if-statements are reorganised accordingly.

6 CONCLUSIONS

This paper set out to investigate platform independent dynamic Java Virtual Machine analysis using the Java Grande Forum benchmark suite as a test case. This type of analysis, of course, does not look in any way at hardware specific issues, such as JIT compilers, interpreter design, memory effects or garbage collection which may all have significant impacts on the eventual running time of a Java program, and is limited in this respect. It has been shown above however that useful information about a Java programs can be extracted at the intermediate representation level, which can be partly used to understand their ultimate behaviour on a specific hardware platform. The technique has also been shown to help in the design of Java to bytecode compilers.

Although the Java to bytecode compiler does not have access to dynamic execution data, it should be able to put the most heavily used local variable into one of the efficient slots most of the time, yet only the *gcj* compiler seems to make a significant attempt at this. A more common optimisation was in the translation of loop constructs, where each successful iteration involves executing two branching instructions, a potential branch if the condition is false and a backward goto (unconditional branch) at the end of the loop for the *pizza*, *gcj* and *borland* compilers, whereas the other compilers combine both of these into a single conditional branch at the end of the loop.

Overall, this study raises questions about the balance of optimisation work between Java compilers and the interpreter component of the JVM. One possibility is that compiler writers are trying to produce as closely as possible the bytecodes produced by the original SUN compiler so as to avoid incompatibility with the runtime bytecode verifier. If this is so, it may explain why various other efficiency improvements have not been used by different compilers.

Clearly, run-time optimisation techniques will always be essential within the JVM, because of both the potential unreliability of the compiler, and the extra information about the run-time architecture available to the JVM. However, it is not obvious that Java compilers are putting much effort into generating efficient bytecode, and it is arguable that the JVM may be bearing an unreasonable part of the burden of performing these optimisations.

Platform independent dynamic analysis has been shown to be a useful tool for the studying the Grande benchmark suite. For Grande applications Java method execution time is shown to be virtually all in the non-API bytecodes of the programs. This is a significant difference from traditional Java applications such as applets or compiler type tools which spend most of the time in the API. Since a Grande application should use large amounts of processing, I/O, network bandwidth or memory, it is interesting to note how little of the API packages are dynamically used by this benchmark suite. Precompiling the API to some native representation therefore will not yield significant speedup.

As would be expected for the programs considered, the applet and awt packages are not used at all as graphics has been removed from the benchmarks. Of major interest is that the math package is not used by the benchmarks which implies either the benchmarks are not representative of numerical programs or the math package is not in fact of much use to such programs which simply use the java.lang.Math class.

REFERENCES

[1] Bull M, Smith L, Westhead M, Henty D and Davey R. *Benchmarking Java Grande Applications*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.

[2] J. Waldron, *Object Oriented Programs and a Stack Based Virtual Machine*, Journal of South African Computer Society, In press.

[3] T.J. Wilkinson, *KAFFE, A Virtual Machine to run Java Code*, <www.kaffe.org> URL last accessed on 20/10/2000

[4] J. Waldron, C. Daly, D. Gray and J. Horgan, *Comparison of Factors Influencing Bytecode Usage in the Java Virtual Machine*, Second International Conference and Exhibition on the Practical Application of Java, Manchester, UK, April 12-14, 2000.

Feedback directed ahead-of-time compilation for embedded Java™ applications

Aldo H. Eisma,
Object Technology International, Inc.
aldo_eisma@oti.com

1 Introduction

To successfully deploy Java applications on embedded devices, both the target applications and the Java Virtual Machine (JVM) required to run the applications need to fit the characteristics and limitations of this class of devices. Due to market pressures, these devices have to be cheap, small and have low power consumption. The most prominent constraints imposed on the JVM and the applications, are:

- The amounts of ROM and RAM available are limited.
- The processing power of the CPU is limited.

The J9 JVM and its supporting SmartLinker application packager in IBM's VisualAge™ Micro Edition [1] have many features that assist application developers to meet these constraints:

1. SmartLinker removes unused classes, methods and fields from an application using an advanced analysis technique called Rapid Type Analysis (RTA) [6].
2. SmartLinker packages the remaining classes, methods and fields in a ROMable image in a compact format (JXE) that can be executed in-place out of ROM by the J9 JVM. The JXE format is far less verbose than the standard Java class file format, and constants are interned. The JXE format also enables a faster and smaller bytecode interpreter.
3. J9 is a portable, componentized Java runtime and set of class libraries that is provided in configurations ranging in size from 300 KB to a complete Java 1.2 configuration.

Although the J9 interpreter is already highly optimized, some applications may still not execute fast enough in interpreted mode, due to the limited processing power of the target CPU. To speed up Java applications further, J9 can employ just-in-time (jit) compilation techniques to compile Java bytecodes into native machine code at runtime. But, for some embedded systems the jit compilation approach may not be appropriate because the JIT compiler

- requires excessive ROM and RAM to operate, and
- generates native code in RAM, while for embedded applications one would like to execute in-place out of ROM.

Therefore, SmartLinker supports ahead-of-time (aot) compilation: after removing unused classes, methods and fields the packager can compile the remaining methods and include the native code into the JXE, replacing the bytecodes otherwise shipped.

The result is a ROMable JXE that can be executed in-place, without needing a jit compiler at run-time. The precompiled JXE can be as much as 5 times faster than the interpreted JXE, but if all methods are precompiled the JXE may also become 4 times larger. This size blow-up may not be acceptable to resource-constrained devices. The question is whether it is possible to precompile only a subset of the methods, thereby saving in precious size, and still have a good performance improvement over the interpreted JXE.

The remainder of this paper describes the results of our investigation into techniques for feedback-directed ahead-of-time compilation to accomplish the following tasks:

™ Java is a trademark of Sun Microsystems, Inc.
™ VisualAge is a trademark of International Business Machines Corporation, Inc.

- Maximize the performance improvement of an application while at the same time minimizing the size of the precompiled JXE.
- Profile a given embedded application to obtain the required dynamic execution profile.

2 Feedback directed ahead-of-time compilation

The idea behind this paper is to support feedback directed ahead-of-time compilation in a future version of VisualAge Micro Edition as follows:

1. An application is packaged for profiling in one or more JXEs, using VisualAge Micro Edition SmartLinker. This could for example be a JXE without any precompiled methods, with all precompiled methods, or a mix.
2. The application is run, one or more times, on the target VisualAge Micro Edition J9 JVM using a feedback-directed ahead-of-time compilation profiling agent hooked into the target JVM.
3. The resulting profiles are retrieved from the target and imported into the development environment.
4. The user specifies a desired performance level to the packager
5. The resulting profiles are merged and analyzed by the packager, and the packager selects a subset of the methods for precompilation to satisfy the constraints.
6. The packager builds the production JXE with the requested performance level.

2.1 Profiling embedded Java applications

A profiling agent for embedded Java applications needs to be deployed in the target JVM, and it needs to be small and fast so that the profiler does not inhibit the target application.

The profiler must be as portable as the J9 JVM itself. This means that the profiler must work independent of the operating system and CPU it is running on (VisualAge Micro Edition supports various operating systems, like for example QNX/Neutrino[2], Linux, WinNT Embedded and more).

To satisfy these, and other, requirements we decided to implement a dedicated profiling agent using the Java Virtual Machine Profiling Interface (JVMPI) [3,4]. JVMPI has designed to support a powerful, flexible and portable mechanism to hook into and probe the JVM.

We have experimented with sample-based profling and instrumented profiling, which will be explained next.

2.1.1 Sample-based profiling

For relatively long running applications, the sampling technique described by Viswanathan and Liang [4] works very well. With the sampling technique the profiler runs in a separate high-priority thread and samples all active threads at a fixed interval. The smallest interval is 1 ms. For each thread in the application, the CPU time used and the current execution point (class, method and line number) are retrieved.

Sample based profiling has the smallest possible impact on the running application, while remaining portable (a technique described in [5] has better real-time characteristics, but is more difficult to port).

The sampling technique as described also works if the underlying operating system does not support measuring CPU time usage per thread, by distributing elapsed time over the runnable threads. We have tested sampling with and without using OS calls to measure CPU time usage per thread on Windows NT and on QNX/Neutrino, all leading to the same results as presented.

2.1.2 Instrumented profiling

With instrumented profiling, JVMPI will report back selected events to the profiling agents, like for example method entry and exit. Potentially, CPU usage can be measured more accurately by recording the CPU time used per thread for each method entry and exit event.

This only makes sense if the granularity of CPU usage measurement is 1 ms or better. Note: on Windows NT the granularity of thread CPU time measurement is 10 ms, and on x86 QNX/Neutrino it is 1 ms.

Event-based profiling slows down the application considerably because a JVMPI event call back occurs for each method entry and exit. In one experiment we attempted to reduce the impact by enabling only method exit events. When using the resulting profiles to precompiling methods of the test application ranked by CPU time, it became clear that the profile based on method exits only was not usable: instead of 99% of the expected performance gain, only 78% was achieved. Furthermore, instrumented profiling potentially unevenly degrades the performance of the application because of cache misses.

2.2 The selected target application

To investigate various metrics and implementation alternatives for feedback directed ahead-of-time compilation, the SmartLinker application itself (which is a Java application) has been selected as target application for profiling for the following reasons:

- It is a real application, and not an artificial benchmark.
- It is a sizeable application and has a long run time (for the given input), thereby eliminating statistical noise.
- It is easy to reproduce the test results using the same sets of inputs, and it runs unattended.
- It has well understood distinct phases that are either I/O bound, CPU bound, creating a lot of objects, doing lots of string processing, etc.
- It has a typical mix of methods implemented in Java and implemented as Java native methods.
- It is a single threaded application that processes its input in a number of sequential phases that can be individually timed, which makes it easier to interpret the results. The phases are: class loading, profile loading, reduction, inlining and JXE generation.

Techniques that do not work for the selected application are not generally useable. Techniques that are successful are most likely to work for many applications.

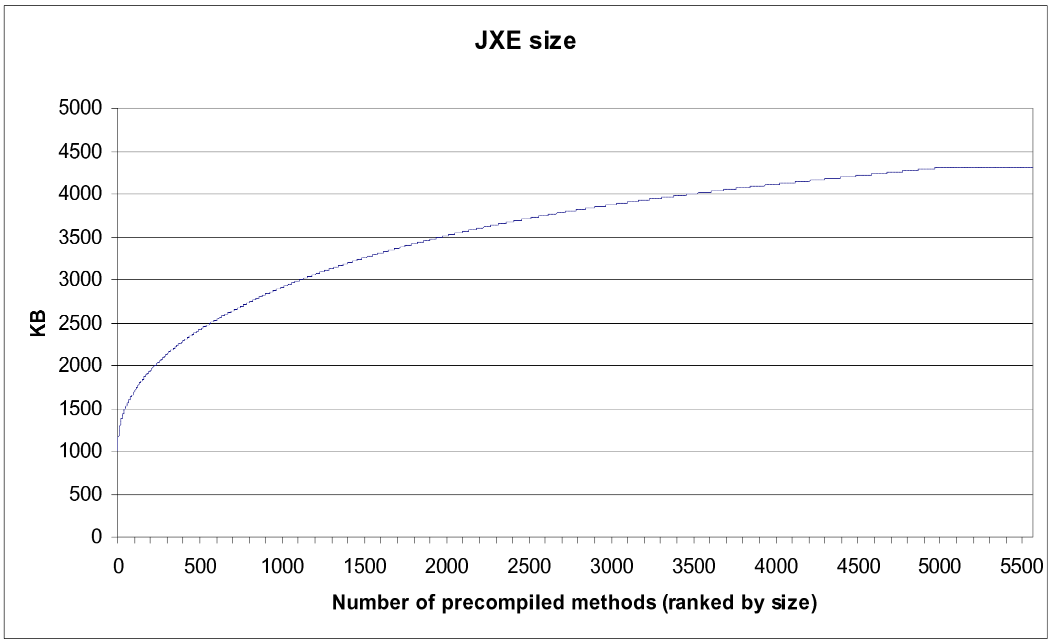
2.3 Base measurements

The following table shows the measured run time (elapsed time in seconds) per phase of a test run without any precompiled methods, and with all methods precompiled. All measurements are done on the VisualAge Micro Edition 1.3 JVM, running on Microsoft Windows NT 4 on an idle 700 MHz Pentium III PC.

% AOT	**Class loading**	**Profile loading**	**Reduce**	**Inline**	**JXE gen**	**Total Time (s)**	**Gain**	**Size (KB)**
0%	26	23	15	274	57	**395**	**0%**	**999**
100%	13	7	3	61	20	**104**	**100%**	**4317**

The table shows that for this application and input data set the maximum possible run time improvement factor is 395/104 = 3.8. As one may expect, the largest run time improvement can be found in the phases without I/O. For the remainder of this paper the gain percentage is defined as the percentage of the maximum possible run time improvement.

The next graph shows how the size of the JXE varies with the number of methods that are precompiled.



The graph shows how the JXE grows from 1 MB to almost 4.5 MB when precompiling successively more methods (sorted by native code size). The JXE is at its maximum size at 5092 methods. The remaining methods are abstract and native methods. The total number of methods is the number of methods in the application, including the class libraries it uses, that remain after all unused methods are removed in the reduction phase.

Now that the test application has been introduced, the question is how to rank the methods in such a manner that the method that contributes to the largest gain when precompiled comes first.

2.4 Precompile by method execution count

The first metric that comes to mind is method execution count. Many jit compilers are triggered to compile a method if the number of times a given method is executed exceeds a threshold. The following table shows the measured execution times resulting from compiling a given percentage of all methods, ranked by highest execution count first.

% AOT	**Class loading**	**Profile loading**	**Reduce**	**Inline**	**JXE gen**	**Total Time (s)**	**Gain**	**Profiled count**
0%	26	23	15	274	57	**395**	**0%**	**0%**
1%	20	21	5	112	42	**200**	**67%**	**96%**
3%	20	20	5	115	43	**203**	**66%**	**98%**
8%	15	15	4	93	31	**158**	**81%**	**99%**
15%	13	7	3	91	25	**139**	**88%**	**100%**
25-100%	13	7	3	61	20	**104**	**100%**	**100%**

The table shows, for example, that if 8% of all methods (ranked by highest execution count first) are precompiled, the execution time is 158 seconds. This corresponds to 81% of the maximum possible gain for this application. The last column (Profiled count) shows the profiled cumulative number of execution counts for the precompiled methods, expressed as percentage of the total number of counts.

From this measurement the following conclusions can be drawn:

- By precompiling 25% of the methods ranked by execution count, the application performs just as well as when all methods are precompiled.
- Even though with 15% of the methods nearly 100% of all calls are covered, one gets only 88% of the maximum possible gain. The explanation is that there are methods with heavy loops that are invoked only a very few times (and often only once).

Can this be improved upon? Yes, as will be shown in the next section.

2.5 Precompile by CPU gain

This metric compares the execution time of every method being interpreted versus being run as native code. To perform this measurement the application is built, and run and profiled for CPU time per method against some input twice: once without any precompiled methods, and once with all methods precompiled. The resulting two profiles are subtracted to obtain the gain in CPU time per method.

The following table shows the measured execution times resulting from compiling a given percentage of all methods, ranked by CPU gain.

% AOT	**Class loading**	**Profile loading**	**Reduce**	**Inline**	**JXE gen**	**Total Time (s)**	**Gain**	**Profiled gain**
0%	26	23	15	274	57	**395**	**0%**	**0%**
1%	18	10	5	65	27	**125**	**93%**	**93%**
3%	16	8	4	63	25	**116**	**96%**	**97%**
8%	14	7	3	62	21	**107**	**99%**	**100%**
15%	13	7	3	65	21	**109**	**98%**	**100%**
25%	14	7	3	64	21	**109**	**98%**	**100%**
33%	14	7	3	62	21	**107**	**99%**	**100%**
43%	14	7	4	70	21	**116**	**96%**	**100%**
52-100%	13	7	3	61	20	**104**	**100%**	**100%**

The table shows, for example, that if 8% of all methods (ranked by profiled CPU gain) are precompiled, the execution time is 107 seconds. This corresponds to 99% of the maximum possible gain for this application. The last column (Profiled gain) shows the profiled cumulative CPU gain for the precompiled methods, expressed as percentage of the total measured CPU gain.

This leads to the following conclusions:

- By precompiling only 8% of the methods ranked by CPU gain, the application performs almost as well as when all methods are precompiled. This is a significant improvement over ranking the methods by execution count.
- There is a good correlation between measured gain and profiled gain, which means that this approach and its implementation is valid (at least for this type of application).

Note: the measured elapsed times for 15% and 25% are unexpectedly slightly higher. Some variation in general has been noticed over different runs, probably as a result of other processes running on the system.

Precompile by CPU gain requires that two versions of the application are built, one with all methods precompiled. This may not be possible on an embedded system because the size of the fully precompiled JXE may exceed the memory limits of the target. Can similar results be obtained using another metric? The next sections show the result for two more metrics.

2.6 Precompile by interpreted CPU time

This metric ranks the methods according to interpreted CPU time. To perform this measurement the application is built without any precompiled methods, run against some input and profiled for CPU time per method.

The following table shows the measured execution times resulting from compiling a given percentage of all methods, ranked by CPU time.

% AOT	**Class loading**	**Profile loading**	**Reduce**	**Inline**	**JXE gen**	**Total Time (s)**	**Gain**	**Profiled cpu**
0%	26	23	15	274	57	**395**	**0%**	**0%**
1%	18	10	5	65	26	**124**	**93%**	**92%**
3%	15	8	4	63	23	**113**	**97%**	**96%**
8%	14	7	3	63	21	**108**	**99%**	**99%**
15%	13	7	3	62	22	**107**	**99%**	**100%**
25-100%	13	7	3	61	20	**104**	**100%**	**100%**

The table shows, for example, that if 8% of all methods (ranked by profiled CPU time) are precompiled, the execution time is 110 seconds. This corresponds to 98% of the maximum possible gain for this application. The last column (Prof.cpu) shows the cumulative CPU time for the precompiled methods, expressed as percentage of the total measured CPU time.

Conclusions:
- By precompiling only 8% of the methods ranked by CPU gain, the application performs almost as well as when all methods are precompiled.
- Just as for precompiling by CPU gain, there is a good correlation between measured gain and profiled gain. This means that this approach and its implementation are also valid.

2.7 Precompile by execution and target call count

The last metric investigated ranks methods according to number of times a method is executed plus the number times target methods are called from a method. The hypothesis is that this metric could compensate the apparent flaw in ranking methods just by execution count. A very busy method that is called only a few times could be calling other methods in a loop.

As the following table shows, this is not a good metric. It is even worse than ranking just by execution count!

% AOT	**Class loading**	**Profile loading**	**Reduce**	**Inline**	**JXE gen**	**Total Time (s)**	**Gain**	**Profiled calls**
0%	26	23	15	274	57	**395**	**0%**	**0%**
1%	25	17	13	256	51	**362**	**11%**	**96%**
2%	25	17	13	257	51	**363**	**11%**	**98%**
5%	21	14	12	252	45	**344**	**18%**	**100%**
10%	21	14	12	252	45	**344**	**18%**	**100%**
20%	21	13	12	258	46	**350**	**15%**	**100%**
30%	19	9	5	163	42	**238**	**54%**	**100%**
40%	19	8	4	159	40	**230**	**57%**	**100%**
50%	19	8	4	158	40	**229**	**57%**	**100%**
60%	16	8	3	62	39	**128**	**92%**	**100%**

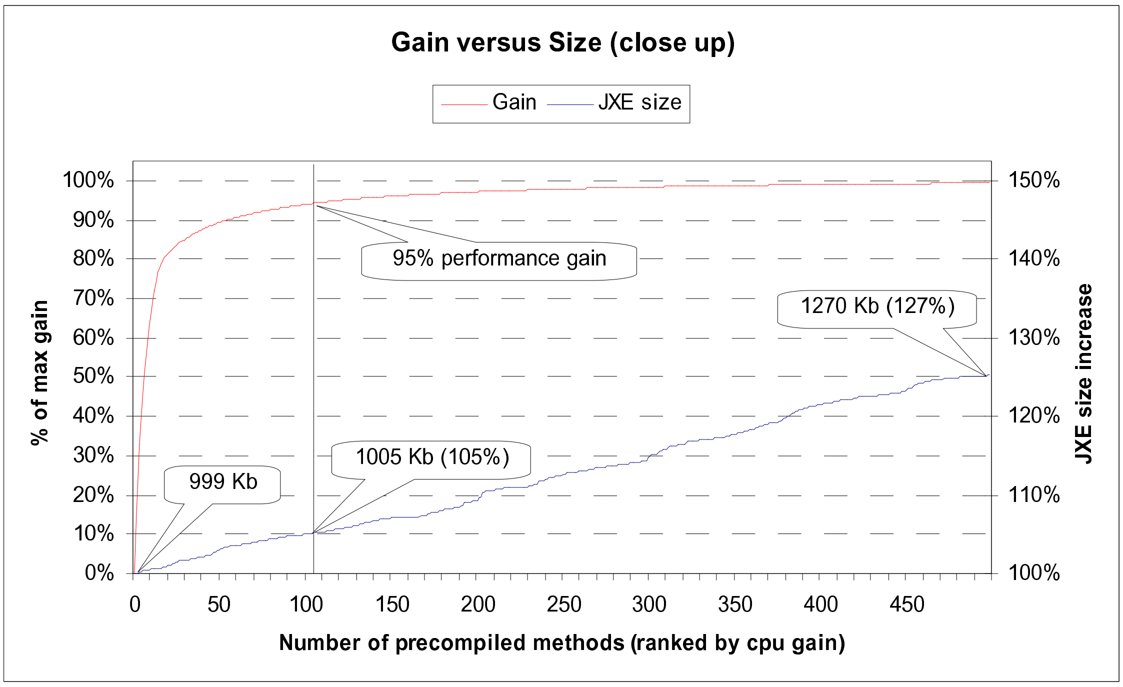
70%	13	8	3	62	26	**112**	**97%**	**100%**
80%	13	8	3	61	21	**106**	**99%**	**100%**
100%	13	7	3	61	20	**104**	**100%**	**100%**

The most likely explanation is that there are methods that are invoked only a very few times that do a considerable amount of processing without frequently calling other methods. Further investigation is required.

2.8 Gain versus size

Since there is a good correlation between gain estimated from the CPU profiles and measured gain, it is useful to have a look at the following graph, showing estimated gain versus JXE size.

The following graph is a close up of precompiling the first 1 to 500 methods, which makes it easier to see that, for example, by precompiling 123 methods (2.2% of all packaged methods) one gets 95% of the maximum possible performance gain. The JXE size is 1049 KB at that point, which is a mere 53 KB (5%) increase over the base size of 996 KB.



To get near the maximum performance gain, less than around 400 (7%) methods need to be precompiled, adding little over 200 KB (20%) to the original size.

3 Conclusion

There is a growing demand from the embedded market for tools that make it possible to develop complex and highly dynamic applications. New tools are on the market that are tailored to the specific needs and limitations of embedded programming in Java, like for example IBM's VisualAge Micro Edition [1]. One of the techniques to improve the performance of an embedded Java application is ahead-of-time compilation of bytecodes into machine code – there is usually no room for a just-in-time compiler on an embedded target. Precompiling all methods in an application may make the ROM image of the embedded Java application 4 times bigger.

It has been demonstrated that by precompiling only a small percentage of all methods, the most CPU-intensive methods, almost the maximum possible performance gain obtainable by precompiling all methods can be achieved. It has been shown that the set of hot methods can be determined by profiling CPU time per method, using simple low-overhead sampling techniques.

More investigation is needed to determine if the results and conclusions extend to other types of applications, like for example heavily multithreaded applications.

4 References

1. IBM VisualAge Micro Edition 1.3, http://www.embedded.oti.com/
2. QNX/Neutrino, http://www.qnx.com/
3. "Java Virtual Machine Profiling Interface (JVMPI)", Java^{TM} 2 SDK, Standard Edition Documentation, Version 1.2.2, Sun Microsytems, Inc., available at http://java.sun.com/products/jdk/1.2/docs/index.html
4. D. Viswanatan, S.Liang, "Java Virtual Machine Profiler Interface", in IBM Systems Journal, Vol. 39, No 1, 2000.
5. John J. Barton, John Whaley, "A Real-Time Performance Visualizer for Java", in Dr. Dobb's Journal, March 1998.
6. F. Tip, C. Laffra, P.F. Sweeny, D. Streeter, "Practical experience with an application extracter for Java", in SIGPLAN Notices 34(10).

Session IV

Run Time Issues

A Fast Java Interpreter

David Gregg[1], M. Anton Ertl[2] and Andreas Krall[2]

[1] Department of Computer Science,
Trinity College, Dublin 2, Ireland.
David.Gregg@cs.tcd.ie
[2] Institut für Computersprachen, TU Wien,
Argentinierstr. 8,A-1040 Wien,
anton@complang.tuwien.ac.at

Abstract. The Java virtual machine (JVM) is usually implemented with an interpreter or just-in-time (JIT) compiler. JITs provide the best performance, but must be substantially rewritten for each architecture they are ported to. Interpreters are easier to develop and maintain, need less memory and can be ported to new architectures with almost no changes. The weakness of interpreters is that they are much slower than JITs. This paper describes work in progress on faster Java interpreters. Our goal is to bring interpreter performance to a new higher level by developing new optimisations for modern computer architectures, and by adapting recent research on compilers to interpreters.

1 Introduction

The Java virtual machine (JVM) is usually implemented by an interpreter or a just-in-time (JIT) compiler. JITs provide the best performance but much of the JIT must be rewritten for each new architecture it is ported to. Interpreters, on the other hand, have huge software engineering advantages. They are smaller and simpler than JITs, which makes them faster to develop, cheaper to maintain, and potentially more reliable. Most importantly, interpreters are portable, and can be recompiled for any architecture with almost no changes.

The problem with existing interpreters is that they run most code much slower than JITs. The goal of our work is to narrow that gap, by creating a highly efficient Java interpreter. If interpreters can be made much faster, they will become suitable for a wide range of applications that currently need a JIT. It would allow those introducing a new architecture to provide reasonable Java performance from day one, rather than spending several months building or modifying a JIT.

This paper describes work in progress on building faster Java interpreters. Why do we think that interpreter performance can be improved? One reason is that interpreters are rarely designed to run well on modern architectures which depend on pipelining, branch prediction and caches. We also believe that many compiler optimisations can be applied to interpreters. Finally, there seems to be a widespread attitude that interpreters are inherantly slow, so there is little point in worrying about performance. The goal of our work is to show that this is not true.

This paper is organised as follows. In section 2 we introduce the main techniques for implementing interpreters. Section 3 describes our work in progress on a fast Java interpreter. Section 4 presents our plans for future optimisations. Finally, in section 5 we draw conclusions.

2 Virtual Machine Interpreters

The interpretation of a virtual machine instruction consists of accessing arguments of the instruction, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction. The most efficient method for dispatching the next VM instruction is direct threading [Bel73]. Instructions are represented by the addresses of the routine that implements them, and instruction dispatch consists of fetching that address and branching to the routine. Unfortunately, direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels.

Fortunately, there is a widely-available language with first-class labels: GNU C (version 2.x); so direct threading can be implemented portably (see 1). If portability to machines without `gcc` is a concern, it is easy to switch between direct threading and ANSI C conforming methods by using macros and conditional compilation.

```
void engine()
{
  static Inst program[] = { &&add /* ... */ };
  Inst *ip; int *sp;

  goto *ip++;

 add:
  sp[1]=sp[0]+sp[1];  sp++;  goto *ip++;
}
```

Fig. 1. Direct threading using GNU C's "labels as values"

Implementors who restrict themselves to ANSI C usually use the giant switch approach (2): VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine; in this method the whole interpreter, including the implementations of all instructions, must be in one function.

Figures 3 and 4 show MIPS assembly code for the two techniques. The execution time penalty of the switch method over direct threading is caused by a range check, by a table lookup, and by the branch to the dispatch routine generated by most compilers.

```
void engine()
{
  static Inst program[] = { add /* ... */ };
  Inst *ip; int *sp;

  for (;;)
    switch (*ip++) {
    case add:
      sp[1]=sp[0]+sp[1];  sp++;  break;
    /* ... */
    }
}
```

Fig. 2. Instruction dispatch using `switch`

```
lw   $2,0($4) #get next inst., $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Fig. 3. Direct threading in MIPS assembly

```
$L2: #for (;;)
 lw    $3,0($6) #$6=instruction pointer
 #nop
 sltu  $2,$8,$3 #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4  #branch delay slot
 sll   $2,$3,2  #multiply by 4
 addu  $2,$2,$7 #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...
$L13: #switch target table
 .word   $L12
 ...
$L12: #add:
 ...
 j   $L2
 #nop
```

Fig. 4. Switch dispatch in assembly

3 A Fast Java Interpreter

We are currently building a fast threaded interpreter for Java. Rather than starting from scratch, we are building the interpreter into an existing JVM. We are currently working with the CACAO [Kra97] JIT-based JVM, but our intention is that it will be possible to plug our interpreter into any existing JVM. It is important to note that we don't interpret Java bytecodes directly. We first translate them to threaded code. In the process, we optimise the code by replacing instructions with complicated run-time behaviour to simpler instructions that can be interpreted more easily. The goal is to put the burden of dealing with complexity in the translator, rather than the inner loop of the interpreter.

The interpreter system consists of three main parts. The *instruction definition* describes the behavior of each VM instruction. The definition for each instruction consists of a specification of the effect on the stack, followed by C code to implement the instruction. Figure 5 shows the definition of `IADD`. The instruction takes two operands from the stack (`iValue1,iValue2`), and places result (`iResult`) on the stack.

```
IADD ( iValue1 iValue2 -- iResult ) 0x60
{
  iResult = iValue1 + iValue2;
}
```

Fig. 5. Definition of `IADD` VM instruction

We have tried to implement the instruction definitions efficiently. For example, in the JVM operands are passed by pushing them onto the stack. These operands become the first local variables in the invoked method. Rather than copy the operands to a new local variable area, we keep local variables and stack in a single common stack, and simply update the frame pointer to point to first parameter on the stack. This complicates the translation of calls and returns, but speeds up parameter handling.

The second part of our interpreter system is the *interpreter generator*. This is a program which takes in an instruction definition, and outputs an interpreter in C which implements the definition. The interpreter generator translates the stack specification into pushes and pop of the stack, and adds code to invoke following instructions. There are a number of advantages of using an interpreter generator rather than writing all code by hand. The error-prone stack manipulation code can be generated automatically. Optimisations easily be applied to all instructions. We can automatically add tracing and profiling to our interpreter. The interpreter generator can also produce a dissassembler, and some of the routines needed by the translator. Specifying the stack manipulation at a more abstract level also makes it easier to change the implementation of the stack. For example, many interpreters keep one or more stack items in registers. It is nice to be able to vary this without changing each instruction specification.

The third part, the *translator* translates the Java byte-code to threaded instructions. In the process of this translation, we rewrite the byte-codes to remove some ineficiencies and make other optimisations more effective. For example, the JVM defines several different load instructions based on the type of data to be loaded. In practice many of these, such as `ALOAD` and `ILOAD` can be mapped to the same threaded instruction. This reduces the number of VM instructions and makes it easier to find common patterns (for instruction combining).

The translator also replaces difficult to interpret instructions with simpler ones. For example, we replace instructions that reference the constant pool, such as `LDC`, with more specific instructions and immediate, in-line arguments. We follow a similar strategy with method field access and method invocation instructions. When a method is first loaded, a stub instruction is placed where its threaded code should be. The first time the method is invoked, this stub instruction is executed. The stub invokes the translator to translate the byte-code to threaded code, and replaces itself with the first instruction of the threaded code. An alternative way to deal with these difficulties is to use *quick instructions*. These are individual VM instructions which replace themselves the first time they are invoked. Unfortunately, they make instruction combining difficult.

One implication of translating the original byte-code is that the design problems we encounter are closer to those in a just-in-time compiler than a traditional interpreter. Translating also requires a small amount of overhead. Translating allows us to speed up and simplify our interpreter enormously, however. Original Java byte-code is not easy to interpret efficiently. Currently, our interpreter runs a small number of Java programs. Once it is fully stable, we will start work on the optimisations described in the next section.

4 Future Directions

Once our basic interpreter is working we will develop new optimsations to make it faster. We will start with those techniques which we believe will give us the most speedup for the least development effort. As out interpreter becomes more efficient, and we become more familiar with the trade-offs we will apply more complicated techniques, whose payoff may be less.

4.1 Reducing Branch Mispredictions

Modern processors depend on correctly predicting the target of branches. In recent work [EG01] we show that efficient VM interpreters contain many difficult-to-predict indirect branches, and that up to 62%–79% of the running time of many interpreters is spent on branch mispredictions. The indirect branches in switch-based interpreters are correctly predicted only about 10% of the time using branch target buffers (the best prediction technique used in current processors). Unless they contain many other inefficiencies, the run time of switch-based interpreters is dominated by indirect branch mispredictions.

The branches in threaded interpreters are correctly predicted about 40% of the time. The reason is that threaded interpreters have a separate indirect branch for each VM instruction, which maps to a separate entry in the processors branch target buffer. We believe that accuracy can be further improved by further increasing the number of indirect branches. For example, conditional branch VM instructions could have two indirect branches rather than one, corresponding to the different directions of the branch. Replicating commonly used instructions will also increase the number of indirect branches. To avoid maintenence problems from multiple versions of instructions, we will modify the interpreter generator to do this automatically. Reducing branch mispredictions is the single most important optimisation for current interpreters, and we expect that it will reduce running time substantially.

Unfortunately, we know of no way to reduce the number of mispredictions in switch-based interpreters. This suggests that it may not be possible to build a particularly fast switch-based interpreter. Our experiments show that the threaded version of existing interpreters is up to twice as fast as the switch-based version of the same interpreter. As we apply optimisations to reduce branch mispredictions, the gap is likely to widen.

4.2 Automatic Instruction Combining

Interpreter overhead can be reduced by combining several Java instructions into one "super" instruction that behaves identically to the sequence, but has the overhead of a single instruction. Previous researchers have used interpreter generators to automatically combine commonly occurring sequences of instructions [Pro95]. Our interpreter generator will also do this, and we plan to experiment with various heuristics for identifying important sequences, based on static and dynamic frequencies.

Another interesting area is more general instruction combining. There is scope for combining not just sequences, but groups of instructions containing branches. For example, it may be that many basic blocks start with a load instruction, so a "branch and conditionally load" instruction may be valuable. Classic optimisations to increase the length of basic blocks, such as tail duplication, may also help instruction combining.

4.3 Run-time instruction combining

Perhaps the most remarkable research on interpreters in recent years is Piumatra and Ricardi's [PR98] technique which copies fragments of executable code to create longer sequences without jumps. This allows instruction combining to take place at run-time, when the program to be interpreted is already known. The technique can be applied to any threaded interpreter, and makes use of GNU C's label variables. They report that it reduces the running the running time of an already fast threaded interpreter by about 30% on a Pentium processor, with almost no reduction in portability or simplicity.

We are keen to evaluate this technique on more modern processors with longer pipelines. It appears to have the potential to allow interpreters to come within striking distance of the performance of JITs. In effect, it allows run-time code generation, but without loss of portability. Clearly, there are many open research questions about the use and effectiveness of this technique.

4.4 Translation to Register VM

The Java VM is a stack machine, which means that all computations are performed on the evaluation stack. For example, to implement the assignment `a = b + c`, one would use the VM instruction sequence `load b; load c; add; store a;`. Register VMs specify their arguments explicitly, so the same statement might be implemented with the instruction `add a, b, c;`. Note that the "registers" in a register VM are usually implemented as an array of memory locations. Register machines have two major advantages. First, as in the example, register machines may need fewer VM instructions to perform a computation. Although the amount of work may be the same, reducing the number of VM instructions reduces the dispatch overhead.

Secondly, register VMs make it much easier to change the order of VM instructions. It is very difficult to reorder stack machine instructions, since all instructions use the stack, so that every instruction depends on the previous one. Register instructions can be reordered provided provided there are no data dependences. Reordering opens new opportunities for instruction combining. We can change the sequence of instructions to better match our "super" instructions.

Register VMs have two important drawbacks. Stack machines implicitly use the top elements of the stack as their operands, whereas register machines must specify the operands. Decoding these operands adds additional overhead. Secondly, there is clearly a cost in translating the JVM stack code to register code. However, for switch-based interpreters, which suffer a branch misprediction for almost every VM instruction they execute, almost anything that reduces the number of executed VM instructions is likely to be of benefit.

4.5 Software Pipelining

Interpreter software pipelining [HATvdW99] reduces the effect of branches on interpreters by moving some of the dispatch code for the next instruction into the current instruction. It was developed for the Phillips Trimdia VLIW processor, which has no branch prediction. Fetching and decoding the next instruction while waiting for the branch for the current one to resolve allowed them to greatly speed up their interpreter.

Software pipelining is also likely to be useful on other processors as a way to reduce the cost of branch mispredictions. Moving the dispatch code into the previous instruction allows much of the dispatch to take place while waiting for the mispredicted branch to resolve. A problem with interpreter software pipelining, is a taken VM branch causes the pipeline to stall. Delayed VM branches can reduce this problem, but require a register VM rather than a stack one, since some VM instruction needs to be moved into the VM branch delay slot. Given that we already plan to experiment with register machines, we plan to investigate the usefulness of delayed VM branches.

4.6 Cache Optimisations

Cache misses have a large and growing effect on the performance of modern computers. Interpreters need little memory, so they work well with caches. Nonetheless, we will investigate how to reduce cache misses further by placing the code to interpret the most frequently executed Java instructions in the same cache lines. This may be very important if, as a result of instruction combining or other optimisations, we greatly increase the size of our VM instruction set.

5 Conclusion

We believe that these optimisations, especially branch prediction ones, will create a very fast Java interpreter. As our interpreter becomes faster, the proportional benefit of other lesser optimisations will increase. For example, if we can reduce running time of our basic interpreter by 50%, then a lesser optimisation that might not be considered worthwhile, since it might give only a 5% speedup on the original interpreter, would give a 10% speedup on the faster one. As interpreters become faster, anything that can eliminate some part of the running time will give a proportionally bigger benefit. Given their simplicity, portability, low memory requirements and reasonable performance, we expect that they will be an attractive alternative to just-in-time compilers for many applications.

References

[Bel73] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[EG01] Anton Ertl and David Gregg. The structure and performance of efficient interpreters. To be submitted in February 2001 to the Europar 2001 European Conference on Parallel Computing., 2001.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[Kra97] Andreak Krall. Cacao - a 64 bit javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11), 1997.

[PR98] Ian Piumatra and Fabio Ricardi. Optimising direct threaded code by selective inlining. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–299. ACM, 1998.

[Pro95] Todd Proebsting. Optimising an ANSI C interpreter with superoperators. In *Proceedings of Principles of Programming Languages (POPL '95)*, pages 322–342, 1995.

ENSEMBLE: A Communication Layer for Embedded Multi-Processor Systems

Sidney Cadot Frits Kuijlman Koen Langendoen Kees van Reeuwijk Henk Sips
Faculty of Information Technology and Systems
Delft University of Technology, The Netherlands
{frits,koen,reeuwijk,sips}@pds.twi.tudelft.nl

Abstract

The ENSEMBLE *communication library exploits overlapping of message aggregation (computation) and DMA transfers (communication) for embedded multi-processor systems. In contrast to traditional communication libraries,* ENSEMBLE *operates on n-dimensional data descriptors that can be used to specify often-occurring data access patterns in n-dimensional arrays. This allows* ENSEMBLE *to setup a three-stage pack-transfer-unpack pipeline, effectively overlapping message aggregation and DMA transfers.* ENSEMBLE *is used to support Spar/Java, a Java-based language with SPMD annotations. Measurements on a TriMedia-based multi-processor system show that* ENSEMBLE *increases performance up to 39% for peer-to-peer communication, and up to 34% for all-to-all communication.*

1. Introduction

The application domain for embedded systems is rapidly expanding. Embedded systems are now also used for multi-media processing (e.g., HDTV) that requires a high level of sustained performance. On the other hand, the life-time of applications is decreasing. This calls for a hybrid solution, in which a general-purpose processor is used for flexibility and multiple embedded processors (DSPs) for high-throughput signal processing. The embedded processors can be configured as a pipeline, or as a processor farm. The latter architecture is often implemented by connecting all processors (host and DSPs) to a shared bus (e.g., PCI) or network (e.g., Ethernet). We refer to this setup as an embedded multi-processor system. In such a system all processors can communicate with each other, which greatly enhances flexibility.

To ease application development for embedded multi-processor systems it is crucial to develop compiler and runtime-system support to handle difficult and error-prone tasks like processor synchronization and data transport between (heterogeneous) processors (e.g., host to DSP). Many parallel programming systems exist ranging from library-based systems (PVM [15], MPI [9]) to language-based systems (HPF [6], CC++ [5], Orca [2]). The latter systems are easy to program because of the parallelizing compiler that hides the complex interface to the parallel hardware. The performance, however, is often compromised because of the layered approach (OS/runtime-system/compiler) taken to achieve portability. The alternative of writing explicitly parallel programs using communication libraries is not very appealing because of the large penalty in development costs – parallel programming at such a low level is difficult and error-prone.

Our approach consists of building a complete tool chain for application development targeting embedded multi-processors. We combine ease-of-programming (compiler) and application performance (efficient DMA-based communication). We support data parallel (SPMD) programming through the Spar/Java language, which is a Java derivative with explicit support for scientific computations [13, 14]. The Spar/Java compiler recognizes annotations for data and code placement, and automatically generates a parallel program with explicit communication. On embedded systems communication is handled by the ENSEMBLE layer, which has been explicitly designed to take advantage of the hardware capabilities (i.e. DMA engines) present in embedded multi-processors. Furthermore we tightly integrated the compiler and ENSEMBLE to overcome the performance penalties (e.g., buffer copies) associated with portable communication libraries.

In this paper we describe the ENSEMBLE communication layer, and evaluate its performance on an Athlon/TriMedia system.

2. ENSEMBLE

Most embedded processors are capable of initiating asynchronous DMA transfers, so that communication and processing can be overlapped. In ENSEMBLE we exploit this feature by overlapping simultaneous DMA transfers

```
final int SZ = 300;

int A[] =
    <$on=(lambda (i) P[(local 0)])$> new int[SZ];
int B[] =
    <$on=(lambda (i) P[(cyclic i)])$> new int[SZ];
int C[] =
    <$on=(lambda (i) P[(cyclic i)])$> new int[SZ];

<$ independent $> foreach( i :- 0:A.length ) {
    A[i] = B[i]*C[i];
}
```

Figure 1. Spar/Java code fragment for inproduct.

```
MESSAGE = MSGBUF;

// all processors: pack outgoing message
for(int i=procno;i<A.length;i+=P)
  *MESSAGE++ = B[i] * C[i];

// all processors: send message to owner
send(MSGBUF,owner(A));

// owner: receive and unpack incoming messages
if (procno==owner(A)) {
    for(int pr=0;pr<P;pr++) {
        receive(MSGBUF,pr);
        MESSAGE=MSGBUF;
        for(i=pr;i<A.length;i+=P)
          A[i] = *MESSAGE++;
    }
}
```

Figure 2. Spar/Java-generated C++ code.

and buffer packing and unpacking as much as possible. The communication performed by data-parallel Spar/Java programs is implicit: whenever a processor references data that resides on another processor, the compiler generates a communication event. The performance of so-called element-wise communication is poor. Therefore, the compiler uses message aggregation to send multiple data elements in a single message. With ENSEMBLE we are able to overlap message aggregation (computation) and communication. This is not possible when the Spar/Java compiler targets traditional message passing libraries like PVM and MPI.

Consider the Spar/Java code fragment in Figure 1. It computes the inproduct of two cyclically distributed arrays (B and C), and stores the result in array A located entirely at processor 0. The Spar/Java compiler generates C++ code with explicit send and receive primitives. Figure 2 shows the generated inproduct C++ code (edited for readability).

The Spar/Java compiler performs a sophisticated analysis to identify opportunities for message aggregation [12]. With the inproduct code, the compiler infers that B and C

```
MESSAGE = MSGBUF;

// all processors: pack outgoing message
for(int i=procno;i<A.length;i+=P)
  *MESSAGE++ = B[i] * C[i];

// all processors: send message to owner
emb_send(owner(A),MSGBUF,MESSAGE-MSGBUF);

// owner: receive and unpack incoming messages
if (procno==owner(A)) {
  for(int pr=0;pr<P;pr++) {
    int stride = P;
    int cnt = A.length/P + (A.length%P > pr ? 1:0);

    emb_recv(pr,sizeof(int),A,pr,1,&stride,&cnt);
  }
}

emb_fence();
```

Figure 3. ENSEMBLE-style C++ code.

are identically distributed, so it can compute the expression B[i]*C[i] at the owning processor and send the result to processor 0. The individual results computed at one processor are aggregated in a single message. Processor 0 processes all incoming messages (including the message sent by itself) by storing the result values in the array A.

Note that a message is completely assembled before being send; likewise, a message is first received before its contents is processed. This setup rules out overlapping message aggregation and communication by ENSEMBLE. Furthermore, the Spar/Java compiler determines in which order the messages are processed. If messages happen to arrive in a different order at runtime, they cannot be processed immediately, but must be buffered. To avoid unnecessary waiting, we would like to process messages on a first-come first-serve basis.

Overlapping message aggregation and communication requires a tight integration: either the compiler must be made communication aware (e.g., address fragmentation for pipelining), or the message passing layer must provide a higher-level interface (e.g., scatter-gather message vectors). With ENSEMBLE we take the latter approach and operate on (n-dimensional) *data descriptors* instead of contiguous buffers. Furthermore, we require that all sends and receives are registered before invoking ENSEMBLE to perform the actual data transfers. This registration-execution mechanism provides ENSEMBLE with the opportunity to (re-) schedule data transfers dynamically to match availability of data (buffers) at source (destination) processors.

We modified the Spar/Java compiler to generate ENSEMBLE-style code as shown in Figure 3. At the sending side nothing seems to have changed, but the actual transmission of the message is delayed until emb_fence is invoked. At the receiving side ENSEMBLE is instructed to unpack messages directly into array `A`: the data must be assigned by cycling through the array with stride `P` starting at index `pr`. The net effect is that the data transmission can be overlapped with the unpacking in any order that the messages arrive.

2.1. Data descriptors

Both source and destination of a data transfer in ENSEMBLE must be specified using 'data descriptors'. These allow a variety of often-occurring data access patterns to be specified as either the source or destination of a transfer.

The data descriptors used in ENSEMBLE are based on the concept of selecting the elements of a one-dimensional array with index-values ranging from a lower-bound L up to an upper-bound U, each time incrementing the index with a stride S. Actually, the specification as used in ENSEMBLE uses a slight modification of this scheme, using a fixed lower bound L=0; instead of the upper bound U the 'element-count' C is used, for which the following holds:

$$C = \left\lfloor \frac{U}{S} \right\rfloor + 1$$

The (S,C) specifications can be nested to allow the specification of complex data access patterns. The lack of lower-bounds is compensated by having a 'base-element index' B, which designates the element in the source array that serves as the starting point for strided packing or unpacking. The advantage of this approach compared to using a full (L,U,S) specification is that it suffices to have one base-index value B per (S,C)-specification list, instead of having one lower-bound value per (L,U,S) specification.

One obvious way in which nested (S,C)-specifications can be used is to traverse several dimensions of a multi-dimensional array. As an example, consider a 3-dimensional array, residing on one processor, that is to be cyclically distributed among three processors (Figure 4). The colors of the layers in the z-direction indicate the destination processors; the sender will need to specify three 'send' operations. Table 1 shows the three data-descriptors needed in specifying the 'send' operations: each of these consists of one first-element offset B, and three (S,C) pairs.

Apart from specifying which elements to use, the base offset and (S,C)-pairs also specify an *order* in which the elements must be traversed. In ENSEMBLE, the first-specified (S_0,C_0) pair denotes the inner loop. It is allowed for the source and destination data-descriptors to describe quite different array-traversals; only the total number of elements

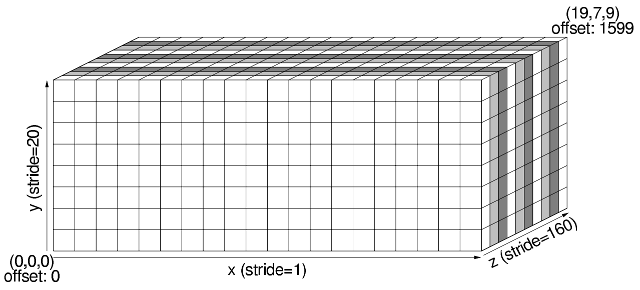


Figure 4. 3D array to be cyclically distributed.

layers	B	(S_0,C_0)	(S_1,C_1)	(S_2,C_2)
white	0	(1,20)	(20,8)	(480,4)
light-gray	160	(1,20)	(20,8)	(480,3)
dark-gray	320	(1,20)	(20,8)	(480,3)

Table 1. Data descriptors for the three targets.

must be equal. This feature can be used in interesting ways, in which the ENSEMBLE layer is effectively used to perform a data-transforming operation. Suppose we have a square 100 × 100 matrix at processor 0, and that we need the transposed contents of this matrix on processor 1. Table 2 shows the way in which this can be accomplished using the ENSEMBLE data-descriptors.

	B	(S_0,C_0)	(S_1,C_1)
sender	0	(1,100)	(100,100)
receiver	0	(100,100)	(1,100)

Table 2. 100x100 matrix transpose descriptors.

For many operations in which standard distributions such as 'block', 'cyclic', and 'block-cyclic' distributions are used, it is possible to express the send- and receive-operations with one or two data-descriptors.

2.2. API

This section describes the small set of primitives provided by the ENSEMBLE layer. The Application Programming Interface (API) comprises ten calls, see Figure 5.

The `emb_init()` call must be called once at startup, before using any other function of the ENSEMBLE layer. The `nproc` parameter specifies the number of processors that should be brought online when bootstrapping the multiprocessor system. The value returned by the `emb_init()` function indicates the processor ID, ranging from 0 to (`nproc`-1), inclusive. When the `emb_init()` call returns, it is guaranteed that the parallel system is online, that the

```
int  emb_init(int argc, char **argv, int nproc);
void emb_done(void);

int  emb_PID(void);
int  emb_NP(void);

void emb_barrier(void);

void emb_send(int dst, int elmSize, void *base,
              int offset, int nstrides,
              int *strides, int *counts);

void emb_send_block(int dst, void *base, int sz);

void emb_recv(int src, int elm_sz, void *base,
              int offset, int nstrides,
              int *strides, int *counts);

void emb_recv_block(int src, void *base, int sz);

void emb_fence(void);
```

Figure 5. ENSEMBLE interface.

program is running on a processor that participates in the parallel system, and that other ENSEMBLE calls may be performed. The function cannot fail; if the bootstrapping process fails within the `emb_init()` call, ENSEMBLE will immediately abort program execution.

The `emb_barrier()` call implements a full barrier, synchronizing all processors. This means exactly the following: upon return from the i-th call to `emb_barrier()`, all processors are guaranteed to have at least entered the i-th call to `emb_barrier()`.

The `emb_send` and `emb_recv` functions are used to register 'send'- and 'receive'-type operations, which will be executed upon the next call to `emb_fence()`. The `dst` and `src` arguments specify the 'peer processor' of the specified operation; a matching call must be executed on that processor. Send-to-self is allowed, and should also have a matching receive-from-self call. The `elm_sz` argument specifies the element size of the array elements, while the `base` argument specifies the array pointer. The `offset` argument specifies the element relative to which all strides will be performed; this is the B value that was discussed in Section 2.1. The `nstrides`, `strides`, and `counts` parameters specify the number of nested strides, and the values S_i and C_i, respectively. These are processed as soon as the registering call is made; there is no need to preserve these arrays until the next `emb_fence()` call.

The `emb_send_block()` and `emb_recv_block()` calls are shorthand functions for sending and receiving contiguous buffers. These are treated exactly like the corresponding `emb_send()` and `emb_recv()` calls, and may be freely mixed.

Invoking the `emb_fence()` starts the actual data transfers involved in the sends and receives registered since the last call completed. The order in which the sends (and receives) are handled is left unspecified, except that sends to the same destination are handled in FIFO order. This provides ENSEMBLE with the freedom to schedule a send as soon as its destination is ready to accept the incoming data (i.e. entered the corresponding `emb_fence()` call). The `emb_fence()` call returns as soon as all pending sends and receives are completed; in case no transfer operations are specified (which is allowed), it will simply fall through.

2.3. Streaming

The goal of ensemble is to overlap message aggregation (computation) and DMA transfers (communication). This calls for a pipelined approach with three stages: packing (at source processor), transmitting (DMA engine at source), and unpacking (at destination processor). To reduce startup costs we use relatively small buffers that are passed down the pipeline; each DMA across the PCI bus transfers 4 KB (or less for the last fragment) of data. These transfer buffers are also the basic unit of flow control between sender and receiver. Flow control is implemented per sender/receiver pair: each node allocates a small number (4) of buffers per possible sender (4×(`nproc`-1) in total). Whenever a buffer is emptied (unpacked) at the destination it is returned to the sender immediately; the receiving processor updates the administration at the sender node. The arrival of a buffer at the receiver is similarly indicated by the sender writing in the receiver's administration. (By appending an 'available' flag at the end of the transfer buffer, the signalling can be piggybacked nearly for free onto the DMA transfer.)

2.3.1. Zero copying

Before describing the (un)packing of data elements into and out of transfer buffers it is important to note that in some cases it is better to avoid the complete pipeline altogether. When the source data elements are stored at consecutive memory locations (e.g., a matrix row) and the locations of the elements at the destination are consecutive also, then the most efficient communication method is a single DMA transfer avoiding two intermediate copies. To exploit zero-copying ENSEMBLE performs a handshake between sender and receiver at the start of each communication to check if both source and destination are contiguous buffers. If so, a single DMA transfer is requested streaming data directly from source memory to destination memory. If only one buffer is contiguous, one copy is saved by directly DMA-ing to/from a transfer buffer. Otherwise, the complete pack-transmit-unpack pipeline is started (two copies).

2.3.2. Dimension reduction

When `emb_send()` or `emb_recv()` calls are performed, the client program (generated by the Spar/Java compiler) passes a list of n stride/count specifiers. To enhance performance ENSEMBLE processes this data description to identify consecutive elements (and dimensions). For example, a matrix row specified as a sequence of *cnt* elements (stride 1, size sz), can be regarded as a single buffer (size *cnt* $\times$ sz). Likewise, a sequence of rows can be collapsed to a single buffer. The steps performed when translating a data descriptor to an internal 'stride-copy state' descriptor recognize such cases. This process is called 'dimension reduction' as it may result in a stride specification with less dimensions than the original specification. The dimension-reduction process as performed in ENSEMBLE comprises the following steps:

1 The 'element size' is included as the very first dimension of the stride-copy descriptor.

2 Next, all other dimensions are added. If possible, an added dimension is combined with the previous dimension. This is possible if the stride S_{i+1} of a dimension to be added is equal to the count C_i of the previous dimension, multiplied by the stride S_i of the previous dimension.

3 When all dimensions have been processed, we drop the inner dimension and use its size as the element size of the resulting descriptor.

These steps assure that the minimal number of dimensions is specified needed to visit the same array elements as the original specification, in the same order. If the number of dimensions is reduced to zero, the data is available as one contiguous buffer (with length 'element size'), indicating that we might engage a zero-copying transfer.

2.3.3. Data (un)packing

During the execution of an `emb_fence` operation, the actual message aggregation takes place. The stride-copy descriptor is used to copy data from the source array into the next transfer buffer. The stride-pack routine can handle arbitrary specifications of memory traversal; however, much effort was put into the efficient operation for the most important case in which the packing code can be written as operations on aligned 4-byte elements (holding primitive types like float and int). The stride-pack code contains a highly optimized unrolled loop for this case achieving a throughput of 150 MB/s on our embedded TriMedia target. To use this highly-optimized code in as many cases as possible we check if the element size (s) is a multiple of four. If so, we effectively introduce a new inner dimension with count $s/4$.

2.4. Communication scheduling

The `emb_fence` routine will perform all registered send and receive operations in the correct (FIFO) order. This is achieved internally by providing each processor with both an incoming and an outgoing queue of registered operations with each processor in the system, including itself. Thus, `nproc`$\times 2$ queues must be managed at each processor. The queues are filled with stride-copy descriptors by the `emb_send()` and `emb_recv()` calls. The communication scheduler loops over all queues trying to advance the pending communication action at the head of each queue until all registered sends and receives have been performed. We constructed the scheduler such that each action on some queue is non-blocking and involves a small amount of work ('pack the next buffer' being the most expensive). This avoids deadlock and ensures fair progress (round robin scheduling); multiple pipelined communications are effectively interleaved.

With each queue we associate a state machine that records where execution will continue on the next round of the scheduler. A communication is initiated by the receiver who posts a request at the sender. The sender polls for the request. When it arrives the sender checks if a zero-copy transfer can be used. If so, it initiates the DMA transfer, and modifies the queue state to check for completion. When a future action detects the completion of the DMA, it notifies the receiver and removes the strided-copy descriptor from the head of the queue. A buffered transfer is handled by checking if a free buffer is available. If not, we yield control back to the scheduler. Eventually a buffer will become available, it is filled by strided-pack, and a DMA transfer is initiated. (The strided-pack routine records in the queue state where to proceed for the next fragment.) When detecting the completion of the DMA in a future action, we do not need to explicitly signal the receiver because of the 'available' flag appended to the buffer, but may continue with allocating, filling and transmitting the next fragment. When the DMA of the last fragment completes we remove the strided-copy descriptor from the head of the queue. When all queues are emptied, `emb_fence` returns control to the caller.

3. Implementation

We implemented ENSEMBLE (and the Spar/Java compiler) on a heterogeneous multi-processor system consisting of one host CPU (AMD Athlon [1]) and three multimedia DSPs (Philips TriMedia [11]) connected by a PCI bus. The Spar/Java compiler generates C++ code, which is then compiled for the Athlon and cross-compiled for the TriMedia. The Athlon executable downloads the TriMedia code on the three embedded processors and initiates execution.

The three TriMedia processors perform the actual parallel computation and return their output to the Athlon.

The AMD Athlon host processor is a high performance, x86-compatible microprocessor. In our system it runs at 700 MHz and is equipped with a 64 KB level-1 data cache, a 512 KB L2-cache, and 256 MB of SDRAM. The specifications of the embedded TriMedia TM1000 processor are more modest: 100 MHz processor, 32 KB instruction cache, 16 KB data cache, and 8 MB of memory.

The Athlon processor is under control of the Linux operating system. Linux supports virtual memory, which complicates the ENSEMBLE implementation for two reasons: 1) a TriMedia addressing Athlon memory uses physical addresses, 2) to avoid Linux from swapping out pages, DMA-ble areas must be 'pinned'. Since the user has no control over the relation between (consecutive) virtual addresses and hardware pages, the DMA-ble area is limited to 4 KB (1 page). This prevents ENSEMBLE from using zero-copy transfers between the Athlon and a TriMedia.

The TriMedia boards are equipped with a DMA engine that operates independently and can be controlled from the embedded ENSEMBLE software running on the TM1000 processor. The caveat, however, is that the DMA engine directly operates on SDRAM without consulting the (copy-back) data cache. Therefore, ENSEMBLE code is cluttered with *cache-copyback* instructions to update memory with cached values, and *cache-invalidate* instructions to avoid stale data in the cache when memory has been updated by the DMA engine.

Another characteristic of our system that has caused much grieve is that for some reason DMA transfers and PIO (programmed I/O) can not be mixed freely. Handshaking by directly writing to another processor's memory occasionally fails because the PIO write does not get through to the destination's memory when contending with concurrent DMA transfers for the PCI bus. This results in data loss and, consequently, in deadlock. We now use DMA for all accesses to remote memory at the expense of additional latency.

Figure 6 shows the transfer rates for different message sizes when using DMA. The rates are measured with a hand-coded program issuing DMA-send/receive requests in a tight loop. Note that 'Put'-type operations are invariably faster than 'get'-type operations. When performing a 'get'-type operation, the requester must wait for it to finish, thereby introducing a synchronization effect with the remote hardware. 'Put'-type operations on the other hand can normally proceed at full-speed, using buffers between the sender and receiver and requiring no end-to-end synchronization. ENSEMBLE therefore uses sender-initiated communications, except for Athlon-to-TriMedia transfers due to the lack of an accessible DMA engine on the Athlon. Additional measurements, including Athlon-to-TriMedia and concurrent DMA transfers, are reported in [4].

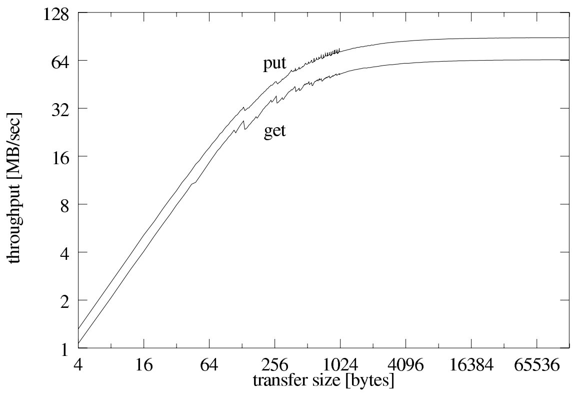


Figure 6. DMA performance (TM to TM).

4. Results

To demonstrate the performance of ENSEMBLE (supporting Spar/Java) we present two benchmarks capturing the often occurring peer-to-peer and all-to-all communication patterns.

4.1. Peer-to-peer

Figure 7 shows the DMA throughput achieved on our hardware for peer-to-peer communication. Processor 1 transmits a number of matrix elements (4-byte floats) to processor 0. For reference the basic DMA put performance from Figure 6 is plotted (top most curve). We measured the performance of ENSEMBLE under the most favorable conditions: the matrix elements are stored consecutively. In this case ENSEMBLE issues a zero-copy DMA transfer. The 'Ensemble zero copy' curve shows that for small data sizes the ENSEMBLE overhead is significant, for example, when sending 64 consecutive floats (256 bytes) ENSEMBLE achieves a throughput of just 11.4 MB/s vs. 46.8 MB/s that is obtained with hand-coded DMA transfers. The overhead is caused by two factors: 1) the preprocessing of data descriptors (dimension reduction, etc.), and 2) the handshaking between sender and receiver to infer the possibility for a zero-copy transfer. The relative impact of the ENSEMBLE overhead rapidly decreases when the data size is increased; beyond 4 KB it is insignificant. We also measured the performance of Spar/Java on top of ENSEMBLE. Spar/Java adds almost no overhead in this simple case, so the curve closely follows the 'Embedded zero copy'; for clarity it is not shown.

To study the performance of ENSEMBLE in less favorable conditions, we forced it to execute the (buffered) pack-transmit-unpack pipeline. The 'Ensemble buffered' curve shows the throughput across all data sizes. The performance drop starts out with a factor 1.6 for 64 elements, increases

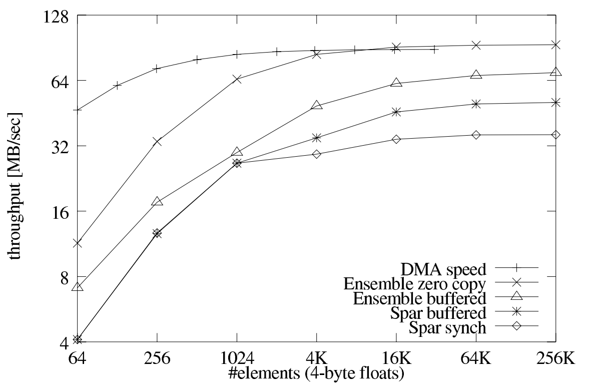


Figure 7. Peer-to-peer performance.

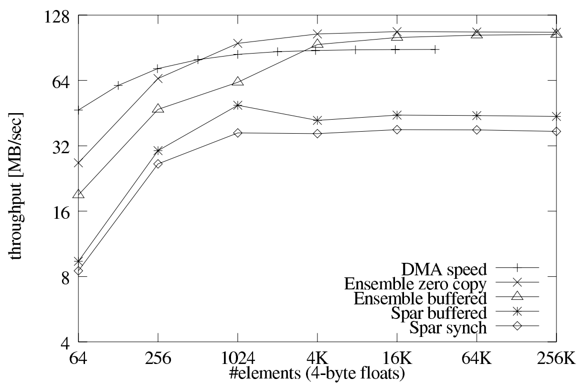


Figure 8. All-to-All performance.

to 2.2 for 1024 elements, and then gradually decreases to 1.3 for 256K elements. This behavior is a consequence of ENSEMBLE using 4 KB buffers for (un)packing; up until 1024 floats (= 4 KB) pipelining is not effective because only one buffer is used. For large data sizes one would expect the pipeline startup costs (pack and unpack of one buffer) to become negligible. This is clearly not the case. The reason is that for large data sizes the pack performance reduces to 70 MB/s due to the need to load (flush) data in (from) the cache; the 150 MB/s rate reported in Section 2.3.3 was obtained for a single 4 KB buffer that together with the source data exactly fits in the data cache. Thus, for large data sizes the bottle-neck in the pack-transmit-unpack pipeline is the pack phase (70 MB/s), not the DMA speed (93 MB/s).

Again we studied the impact of Spar/Java. We measured the performance of a simple test program that assigns a cyclically distributed array (residing on both processor) to an array that is completely local to processor 0. The resulting curve is labeled 'Spar buffered'. Note that the throughput is considerably lower than the corresponding 'Ensemble buffered' curve. The Spar/Java compiler is not the cause of this performance drop (i.e. it does not introduce additional copies), but again the performance of the strided-pack routine. Since the data is cyclically distributed, the array elements at processor 1 are laid out in memory with stride 1 (4 bytes). When packing the elements only 50% of the data in each cache line is used. The additional loads to fill the cache degrade the packing rate from 70 MB/s to 51 MB/s.

To measure the benefits of pipelining we ran the same Spar/Java test program on ENSEMBLE in synchronous mode; ENSEMBLE waits for each DMA to complete before continuing with other work (e.g., packing). This effectively simulates the Spar/Java compiler doing all message aggregation before invoking a traditional communication subsystem. Comparing the 'Spar synch' and 'Spar buffered' curves in Figure 7 shows that pipelining is effective. For example, when communicating 64K elements pipelining raises the throughput from 35.9 to 49.9 MB/s, an increase of 39%. Again we can see the effect of using 4 KB buffers: up until 1024 floats the complete data set fits into a single buffer so no performance is gained.

4.2. All-to-all

We repeated the measurements for the all-to-all communication pattern, in which each processor sends data to all others (including itself). We do not broadcast the data, but send separate messages to individual processors. The performance results are shown in Figure 8. For each data size, six messages of that size are transferred across the bus.

In comparison to the peer-to-peer pattern, the throughput obtained for all-to-all communications are higher (except for 'Spar buffered' with large data sets). ENSEMBLE (both zero copy and buffered) even exceeds the raw DMA speed. The reason is that concurrent DMA transfers issued by multiple processor do better utilize the capacity of the PCI bus (132 MB/s). The highest throughput is achieved by 'Ensemble zero copy': 107.4 MB/s. The performance of 'Ensemble buffered' (104.2 MB/s) approaches the zero-copy throughput closely for large data sizes. This shows that the packing speed (70 MB/s) is not the bottleneck in the pipeline as it was in the peer-to-peer case. Again this is a consequence of performing multiple transfers in parallel: 3×70 MB/s $>$ 107.4 MB/s.

At the Spar/Java level the performance improvement over peer-to-peer communication is only observed for small data sizes ($<$ 4K floats). For large data sizes the performance of 'Spar buffered' even decreases below the 49.2 MB/s achieved with a data size of 1024 elements. The reason is (again) the data layout. The Spar/Java all-to-all test program assigns a cyclically distributed array to a replicated array. Since three processors are involved (against two for peer-to-peer) the cache lines holding the source data are used even less efficiently (33%) than with peer-to-peer

(50%). The end result is that the buffered all-to-all throughput for 256K elements (43.7 MB/s) is less than the buffered peer-to-peer throughput (50.7 MB/s). In the non-pipelining case ('Spar synch') all-to-all performs slightly better than peer-to-peer: 37.3 MB/s versus 36.0 MB/s.

The effects of overlapping message aggregation and data transfer ('Spar buffered' versus 'Spar synch') for all-to-all are different than for peer-to-peer. With peer-to-peer no difference was seen for small data sizes because of the 4 KB transfer buffers. With all-to-all, however, pipelining pays off for all data sizes. The reason is that even if the data fits into one transfer buffer, the time waited for completion of the DMA transfer can be used to pack a message destined for another processor. The largest gain is obtained for 1024 floats: 'Spar buffered' achieves a 34% increase over 'Spar synch'. For large data sizes the poor cache utilization decrease the benefit of pipelining: just a 17% increase for 256K elements.

We plan to enhance our compiler to store all local data of a distributed array in one contiguous buffer. This reduces the memory footprint of a distributed array, and allows the cache to be used more efficiently. This will raise packing speeds considerably and increase the relative impact of pipelining. A potential disadvantage is that every array access requires an additional global-to-local offset translation, but as we describe in [12] this calculation can usually be lifted out of loops.

5. Related work

Overlapping computation with communication is a well known concept; many message passing systems, including MPI, provide asynchronous send (and receive) primitives. Making effective use of these primitives is often the task of the programmer, whereas we use a compiler-based approach. When sending long messages, manual fragmentation and assembly to implement a pipeline is cumbersome. Therefore, thin message passing layers for fast networks like Fast Messages for Myrinet [7], provide a streaming interface where a message may be presented as many small parts; each part is written into the stream using a separate call. Although this efficiently supports scatter/gather message vectors used in many protocol stacks, the function call overhead is prohibitive for sending strided data like Spar/Java requires.

In the way it is used, ENSEMBLE resembles irregular communication libraries such as CHAOS [10]. In both cases a list of communications that must be done is prepared, and upon execution of this list the library is free to choose the optimal approach for the particular communication pattern. However, ENSEMBLE was designed to implement communication of regularly strided blocks efficiently, not irregularly distributed single elements. Moreover, libraries such as CHAOS are usually implemented using standard communication libraries, and therefore do not exploit the possibility of overlapping DMA and message gather and scatter to the extend that ENSEMBLE does.

The embedded multi-processor systems that ENSEMBLE targets bear great resemblance with modern SMPs, which include programmable network interfaces with DMA functionality. Making DMA available to the user in the context of general-purpose SMPs is difficult because of the kernel barrier. Several techniques have been proposed to deal with this issue [3, 8]. Since embedded processors do not support multi-tasking, no true kernel is needed making the design of ENSEMBLE much simpler.

6. Conclusions

Embedded systems supporting multi-media must be flexible to support different applications and high-performance to support the signal processing involved. An embedded multi-processor system consisting of a general-purpose CPU and a number of dedicated DSPs connected by a shared bus is a suitable hardware architecture. To ease application development for embedded multi-processor systems we have developed a compiler (for Spar/Java) and communication layer (ENSEMBLE) that automatically take care of difficult and error-prone tasks like processor synchronization and data transport between heterogeneous processors.

The interface between the Spar/Java compiler and ENSEMBLE is designed such that the hardware capabilities of embedded systems can be exploited. In particular, ENSEMBLE exploits DMA engines to overlap message aggregation (computation) and communication. Spar/Java supports the SPMD computational model and generates rendez-vous style communication with explicit send and receive primitives. In contrast to traditional communication libraries, ENSEMBLE operates on n-dimensional data descriptors that can be used to specify often-occurring data access patterns in n-dimensional arrays. This allows ENSEMBLE to setup a three-stage pack-transfer-unpack pipeline, effectively overlapping message aggregation and DMA transfers. If source and/or destination elements are laid out contiguously in memory, ENSEMBLE (partly) skips the pipeline to bypass intermediate buffers (zero-copy transfer) increasing throughput rates.

We implemented Spar/Java and ENSEMBLE on a embedded multi-processor system consisting of an Athlon host processor and three TriMedia TM1000 multimedia processors. Performance measurements show that ENSEMBLE adds little overhead to the raw DMA speed. When supporting Spar/Java, the actual communication speed largely depends on the layout of the data elements involved; the pack and unpack routines are sensitive to how well the cache handles strided accesses. We determined that overlapping message aggregation and communication increases performance up to 39% for peer-to-peer communication, and up to 34% for all-to-all communication. We anticipate an even larger benefit when the Spar/Java compiler will implement shrinking to store distributed arrays compactly.

Acknowledgements

The design and implementation of ENSEMBLE was carried out within the JOSES project funded by Esprit (LTR Project #28198).

References

[1] AMD Athlon^{TM} Processor Technical Brief, Dec. 1999.

[2] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[3] M. Blumrich, C. Dubnicki, E. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *HPCA-2*, pages 154–165, San Jose, CA, Feb. 1996.

[4] S. Cadot, K. Langendoen, H. Sips, and C. van Reeuwijk. Implementation of H-PAM, ENSEMBLE: A communication layer for the embedded heterogeneous multi-processor target of the Spar compiler. JOSES deliverable 5.2.2/1, DUT-8105-ENSEMBLE, Oct. 2000.

[5] K. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT press, 1993.

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, Feb. 1997.

[7] M. Lauria, S. Pakin, and A. Chien. Efficient layering for high speed communication: Fast Messages 2.x. In *7th High Perf. Distributed Computing Conf. (HPDC7)*, Chicago, Illinois, July 1998.

[8] E. Markatos and M. Katevenis. User-level DMA without operating system kernel modification. In *HPCA-3*, pages 322– 331, San Antonio, TX, Feb. 1997.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, final report v1.0 edition, Apr. 1994.

[10] S. Mukherjee, S. Sharmann, M. Hill, J. .Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed memory machines. In *PPoPP 95*, pages 68–79, Santa Barbara, CA, July 1995.

[11] Philips Electronics, TriMedia Product Group. *TM1000 Preliminary Data Book*, 1997.

[12] C. van Reeuwijk, W. Denissen, H. Sips, and E. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, Sept. 1996.

[13] C. van Reeuwijk, F. Kuijlman, and H. Sips. Extending Java with constructs for scientific computation. PDS Technical Report PDS-2001-001, Delft University of Technology, Feb. 2001. www.pds.twi.tudelft.nl/reports/2001/PDS-2001-001, accepted for publication at the Joint ACM Java Grande/ISCOPE 2001 Conference.

[14] Spar/Java compiler website . http://www.pds.twi.tudelft.nl/timber.

[15] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.