Code Generation for an Application-Specific VLIW Processor With Clustered, Addressable Register Files

Ivan Llopard, Albert Cohen, Christian Fabre, Jérôme Martin, Henri-Pierre Charles, Christian Bernard

To cite this version:

Ivan Llopard, Albert Cohen, Christian Fabre, Jérôme Martin, Henri-Pierre Charles, et al.. Code Generation for an Application-Specific VLIW Processor With Clustered, Addressable Register Files. ODES'13 - 10th Workshop on Optimizations for DSP and Embedded Systems, associated with CGO, Feb 2013, Shenzhen, China. ACM, pp.11-19, 2013, <10.1145/2443608.2443612>. <hal-00911896>

HAL Id: hal-00911896
https://hal.inria.fr/hal-00911896

Submitted on 1 Dec 2013

Code Generation for an Application-Specific VLIW Processor With Clustered, Addressable Register Files

Ivan Llopard
INRIA and CEA Grenoble, France
ivan.llopard@inria.fr

Albert Cohen
INRIA and ENS Paris, France
albert.cohen@inria.fr

Christian Fabre
CEA Grenoble, France
christian.fabre1@cea.fr

Jérôme Martin
CEA Grenoble, France
jerome.martin@cea.fr

Henri-Pierre Charles
CEA Grenoble, France
henri-pierre.charles@cea.fr

Christian Bernard
CEA Grenoble, France
christian.bernard@cea.fr

ABSTRACT

Modern compilers integrate recent advances in compiler construction, intermediate representations, algorithms and programming language front-ends. Yet code generation for application-specific architectures benefits only marginally from this trend, as most of the effort is oriented towards popular general-purpose architectures. Historically, non-orthogonal architectures have relied on custom compiler technologies, some retargettable, but largely decoupled from the evolution of mainstream tool flows.

Very Long Instruction Word (VLIW) architectures have introduced a variety of interesting problems such as clusterization, packetization or bundling, instruction scheduling for exposed pipelines, long delay slots, software pipelining, etc. These have been addressed in the literature, with a focus on the exploitation of Instruction Level Parallelism (ILP). While these are well known solutions already embedded into existing compilers, they rely on common hardware functionalities that are expected to be present in a fairly large subset of VLIW architectures.

This paper presents our work on back-end compiler for Mephisto, a high performance low-power application-specific processor, based on LLVM. Mephisto is specialized enough to challenge established code generation solutions for VLIW and DSP processors, calling for an innovative compilation flow. Conversely, even though Mephisto might be seen a somewhat exotic processor, its hardware characteristics such as addressable register files benefit from existing analyses and transformations in LLVM. We describe our model of the Mephisto architecture, the difficulties we encountered, and the associated compilation methods, some of them new and specific to Mephisto.

International Symposium on Code Generation and Optimization (CGO'13)
Copyright 2013 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Keywords

Clustered VLIW, address generation, LLVM, back-end compiler

1. INTRODUCTION

Modern telecommunication algorithms for Software Defined Radio (SDR) demand high performance computing and flexibility along with hard real-time requirements. Such algorithms are embedded into high-end mobile devices where low power consumption is of primary concern. To meet all these needs, a VLIW-based Application-Specific Instruction set Processor (ASIP), *Mephisto*, has been proposed by Bernard et al. [3]. Mephisto is part of a Network-on-Chip (NoC) based digital baseband for telecommunication protocols such as Long Term Evolution (LTE) [4]. It implements a powerful instruction set specifically designed to provide high-throughput complex number processing. In [3], a code generation framework that represents the instruction set using C++ classes is proposed. This framework has a very low level model of the architecture, directly exposing Mephisto complexity to the programmer. In this paper, we propose a full compilation process based on LLVM, an open source compiler infrastructure [14]. LLVM has already been ported to VLIW architectures, e.g. the Hexagon processor from Qualcomm. However, to our knowledge, it has not been faced to highly constrained VLIW application-specific processors. We also propose new algorithms compatible with generic compilation flows like LLVM, and harnessing the partitioned memory architecture and clustered, addressable register files of Mephisto.

The next section gives an overview of Mephisto, hinting at the implied issues on the compiler side. Section 3 introduces LLVM and presents some of its limitations regarding the implementation of VLIW and DSP back-ends. Section 4 describes the dedicated C programming interface of Mephisto. Our main contributions are discussed in Section 5, highlighting the handling of isolated register files and allocation for addressable register files. Finally, Sections 6 and 7 present some preliminary results and related research work.

2. MEPHISTO ARCHITECTURE

Mephisto is a high performance low-power application-specific processor. It is based on a VLIW architecture aiming high throughput complex number processing with accumulation-based and saturated fixed-point arithmetic. It provides different hardware data paths for memory addresses (or register file addresses) and actual algorithm results. At the bottom of figure 1 are listed some of its operators. They are extensively enumerated here: two Multiply-and-Accumulate (MAC), two CORDIC implementations for Cartesian-polar conversions, a divider, a compare and select unit for fast comparisons, input and output FIFOs. These operators are available for data processing but not intended to be used for address computing. On top of figure 1, there is an overview of the address computing system. The pointer arithmetic is computed by multiple *Address Generators*, one for each memory (*AGma* and *AGmb*) and three for the data register file (*AGa*, *AGb* and *AGw*). The data register file can be indirectly addressed through the use of these address generators. It has two output ports, *ra* and *rb*, that can be loaded with dedicated load instructions. Each output port has an associated address generator, *AGa* and *AGb* for ra and rb, respectively. Addresses for write accesses are taken from *AGw*. The data register file contains 64 slots with 32 bits wide registers (two 16 bits paired registers) while each address generator contains 8, 10 bits wide register pointers. Further details are given in the following sections.

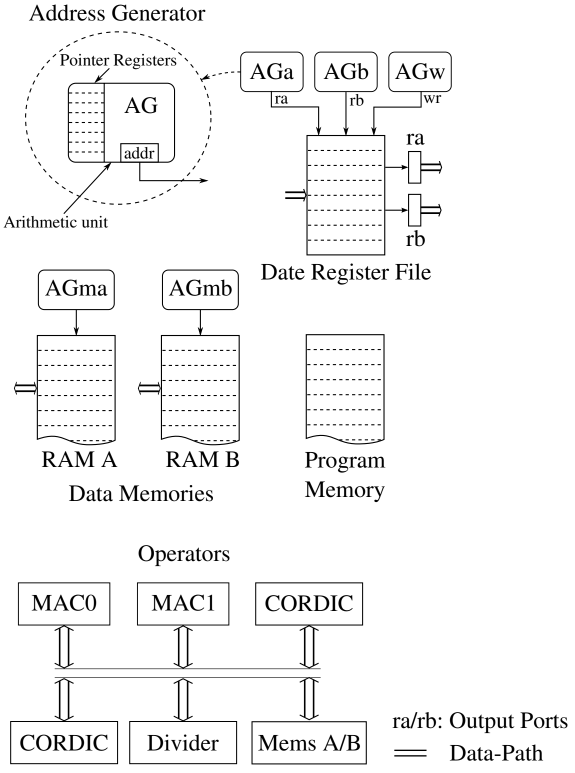

Figure 1: Mephisto: organization of data and address computing units.

2.1 Register files

Mephisto has strong partitioned register files, i.e. inter register file communication is partial or not supported at all. Similar to other register pointer architectures proposed in [20, 23], Mephisto implements *Pointer and Data registers* as well but in a more segregated fashion.

- Data register file: It is read by means of two output ports or registers, *ra* and *rb*.
- Pma and Pmb: Pointer registers that are used to address memories A and B. Contained in address generators AGma and AGmb, respectively.
- Pra and Prb: Pointer registers that are used to address the data register file for read accesses, associated to its output ports, *ra* and *rb* respectively.
- Pw: Pointer registers used to address the data register file for write accesses.

Figure 2 shows all register files and valid transfers between them with directed arrows. We can see that copies between register pointers and from them to data registers are not supported.

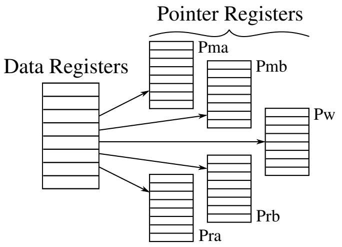

Figure 2: Register files and valid transfers between them.

Data and pointer registers have its own arithmetic units, the multiply-and-accumulate and the address generator unit, respectively. They are introduced in the next sections.

2.2 Multiply and accumulate unit

Mephisto contains two Multiply and Accumulate units, called MAC0 and MAC1, with two accumulators each one. The MAC can perform multiple pipelined operations: two multiplications, addition, accumulation, right arithmetic shifting and saturation (see figure 3). Two output ports connect the data registers to the MAC's. Its structure is optimized for complex number multiplications. Note that a complex multiplication, described hereafter, has four scalar multiplications, one addition and one subtraction

$$Re\{a \times b\} = Re\{a\} \times Re\{b\} - Im\{a\} \times Im\{b\}$$

$$Im\{a \times b\} = Re\{a\} \times Im\{b\} + Im\{a\} \times Re\{b\}$$

where a and b are two complex numbers and $Re\{a\}$ and $Im\{a\}$ denote the real and imaginary part of a, respectively. Therefore, Mephisto is able to perform one complex multiplication per cycle.

In despite of its flexibility, the compiler needs to handle multiple constraints regarding its internal accumulator registers, which are inherent to the MAC behavior:

1. Copies between accumulators are not allowed.
2. Accumulators cannot be spilled. They serve strictly to hold intermediate results.
3. Accumulation is performed on a single accumulator, i.e. two different registers cannot be specified as source and destination of an accumulation operation.

Section 5.3 details how these constraints has been addressed.

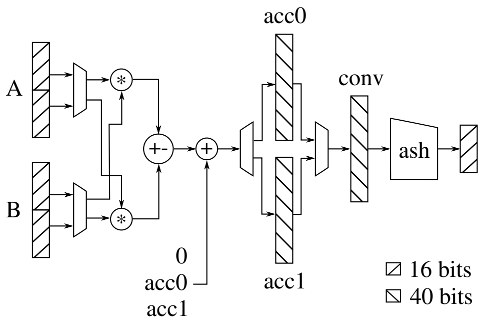

Figure 3: MAC functional representation.

2.3 Address generators

Each pointer register file is contained inside a functional block called *Address Generator*. It is designed to provide address computing support to access data register file and memories. It is composed by a dedicated arithmetic block, which is able to perform modulo n additions, bit shuffling, etc. This specialized computational unit homogeneity imply additional compiler-side work to identify operations for data processing from those for addresses computing. Furthermore, operations for address computing needs to be allocated to the corresponding address generator.

This identification mechanism is called *contextualization* and it is discussed in section 5.1.

2.4 Zero-cost loops

Mephisto provides a limited set of instructions for control flow. It comes from the fact that most of the ported algorithms for Software Defined Radio (SDR) have predictable paths of execution. Unconditional branches are supported as well as conditional ones but within certain functional specific goals. Outside this context, conditionally branching is considered unusual and induces a performance penalty. As other existing DSP's, Mephisto implements zero-overhead loops. Section 5.2 gives detailed information about the hardware loop construction pass.

3. LLVM COMPILER

LLVM is an open source compiler infrastructure composed by a collection of reusable state-of-the-art compiler technologies [14]. Its highly modular design, in form of well-defined libraries, allows it to be used in multiple environments that aims code analysis, optimization and/or generation. LLVM has originally started as a research project at the University of Illinois. Nowadays, it has become a cutting-edge production compiler capable of generating code for a variety of targets, such as x86, ARM, MIPS, PowerPC and more.

3.1 Distance between the semantics of application-specific VLIWs and LLVM

Aside from super-scalar compiler related support, representations of common VLIW functionalities have been added to the code base mainly motivated by the integration of Hexagon back-end, a DSP processor designed by Qualcomm.[1] Thus, recent versions of LLVM include a fairly amount of traditional VLIW needs, e.g. representation of instruction bundles, target-independent packetizer, deterministic finite automaton (DFA) for packet validation. Nevertheless, there is work to be done in order to have a good base of widely known optimizations [9]. On the other hand, a lot of effort has been put to improve existing analysis and optimizations to enhance code generation for VLIW's. For example, the instruction scheduler uses alias information at machine code level to aggressively build the data dependency graph, thus achieving better code schedules. New bundle-aware scheduling heuristics, well suited for these architectures, have been also added to LLVM [13].

MAC-based arithmetic implies new semantics that are normally handled at compiler back-end side, e.g. accumulations. Depending on the MAC internal structure, this approach might require more sophisticated implementations. Front-end support that takes into account this kind of arithmetic would be desirable here to relieve the instruction selector from complex pattern matching logic. Extensions specified by the technical report ISO/IEC 18037 [11] concerning saturated fixed point operations are not yet supported by *clang*, the default C front-end of LLVM. To our knowledge, only address spaces are integrated and fully supported. This constraint has led us to the insertion of new target-specific intrinsic functions to provide explicit saturation semantics.

4. PROGRAMMING INTERFACE

Mephisto is integrated into a data-flow based environment. Its common interface to the external processing units is composed by input and output FIFOs. Two specific functions allow the programmer to read and write them, *readf* and *writef*. The accumulation example of listing 1 performs a simple mathematical operation that can be defined as:

$$R1 = \sum_{i=1}^{400} \bar{u_i} \times v_i \tag{1}$$

where u_i and v_i are two complex numbers and $\bar{v}$ represents the conjugate of v. The values of u and v are interleaved in the input data stream and read one by one. Finally, the result of (1) is sent to the output FIFO.

Listing 1: accumulation.c – Accumulation example

```
// int2 = LLVM vector type <2 x int>
int2 D1, D2, R1;
long long acc0 = 0, acc1 = 0;
for (int i=0; i<400; i++) {
  // read FIFO
  D1 = readf();
  D2 = readf();
  // conj(D1) * D2
  acc0 += D1.x * D2.x + D1.y * D2.y;
  acc1 += D1.x * D2.y − D1.y * D2.x;
}
// arithmetic shift and saturation
R1.x = asrsat(acc0, 18);
R1.y = asrsat(acc1, 18);
// write FIFO
writef(R1);
```

As explained in section 3.1, saturation is introduced with intrinsic functions: *asrsat*. An arithmetical right shift of the accumulator is performed, 18 bits for precision reasons, before its value get saturated to 16 bits. The approach exposes several target-specific behaviors offering a good trade-off between programmability and control on the generated code.

[1] https://developer.qualcomm.com/hexagon-processor

However, the former needs to be improved and high-level language constructs are required.

5. WORK DESCRIPTION

The general overview of the compilation flow is shown in figure 4. It describes the most important passes of our code generation pipeline starting from "target-independent" passes, till instruction selection, and ending at assembler code emission. All along the section 2, we have introduced the principal constraints imposed by the architecture. The following sections further develop those constraints and explain how we have addressed them.

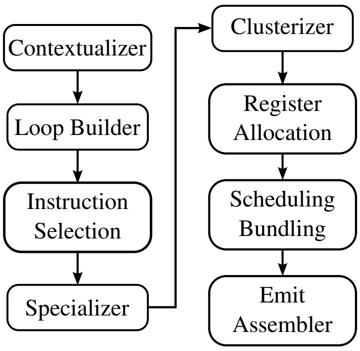

Figure 4: General overview of the code generation flow for Mephisto

5.1 Contextualization

As introduced in section 2.3, Mephisto implements five address generators. They proposes distinct computing contexts for address calculation that are intended to access different memories. Thus, operations cannot be arbitrarily affected to operators without knowing before to which context they belong to. One important context element from the C language is its type system. However, there is no direct relation to the execution contexts imposed by address generators. In other words, address and data can be either explicitly (casts) or implicitly (array indexation) mixed. The greedy instruction selector of LLVM uses a two dimensional approach, *i.e.* machine instructions are mainly selected following the operation type and its associated data types. Nevertheless, the aforementioned constraint shows that the granularity of the instruction selector is not precise enough and instructions need to be contextualized in order to be correctly selected. For technical reasons, we have decided to not modify it but rather to fix its output.

Contextualization can be done at different instruction representation levels. The higher in the CodeGen pipeline is executed, the more optimization phases we benefit from.[2]

A pass at IR level has been implemented to separate data dependent instructions used to compute addresses belonging to distinct memories and even different access types, e.g. when addressing data registers. Given the Directed Acyclic Graph (DAG) $G = (I, E)$ of data-dependent instructions, where I is the set of instructions, it walks G starting from a random load or store i_0, save its address space number and look if it finds a cross address space usage at some other data-related load or store i_n. The data path $P = (i_1, i_{n-1}) \rightarrow P'$ is then cloned and i_n, the faulty memory access, is modified to use i'_{n-1} as its new operand. Cloned instructions are remembered and reused if necessary. The process is repeated until all conflicting paths are eliminated. As an example, listings 2 and 3 show a simple C code and its corresponding IR output.

Listing 2: ireg.c – Array mapped to the data register file

```
static int2 __attribute__((address_space(257)))
  reg[32];
for (int index = 0; index<16; index++) {
  reg[index] = readf();
  reg[index+16] = readf();
  reg[index] = reg[index] + reg[index+16];
}
```

Listing 3: IR output of ireg.c

```
for.body:
  %index.08 = phi i16 [ 0, %entry ],
     [ %inc, %for.body ]
   ...
  %arrayidx = getelementptr inbounds
     [32 x <2 x i16>] addrspace(257)* @ireg.reg,
     i16 0, i16 %index.08
   store <2 x i16> %0, <2 x i16>
     addrspace(257)* %arrayidx
  %add = add nsw i16 %index.08, 16
  %arrayidx2 = getelementptr inbounds
     [32 x <2 x i16>] addrspace(257)* @ireg.reg,
     i16 0, i16 %add
   store <2 x i16> %1, <2 x i16>
     addrspace(257)* %arrayidx2
  %2 = load <2 x i16> addrspace(257)* %arrayidx
  ; def %add6
   store <2 x i16> %add6, <2 x i16>
     addrspace(257)* %arrayidx
  %inc = add nsw i16 %index.08, 1; IV
   ...
```

Here, we have an array mapped to the data register file for which three stores and one load appear in the IR output. Following Mephisto's specifications, load and stores from/to the data register file must use register pointers from Pra/b and Pw, respectively, as well as its associated operators. Figure 5 shows the data relationship between the address computing of load and stores. Such relationship must be broken to have independent address computing operations between read and write memory accesses.

Picking the load node, it finds the path $(3, 2)$ first and clone it. Then it discovers $(3, 4, 5, 6, 7, 8, 9)$ and clone it, reusing already cloned nodes such as *arrayidx'*, breaking the data dependency. The conflict resolution is also applied to address computing regarding memories A and B.

The IR-side contextualizer enables high level optimizations but it is not used to guide the instruction selector. Because LLVM preserves the SSA form at machine level code, the contextualizer was easily ported to handle machine instructions, transforming generic selected instructions to its context-based versions.

5.2 Zero-cost loop construction

Mephisto provides zero-cost loops, i.e. a pair of instructions that hide loop counter decrements and verification for

[2] In particular the Length Strength Reduce (LSR) optimization pass

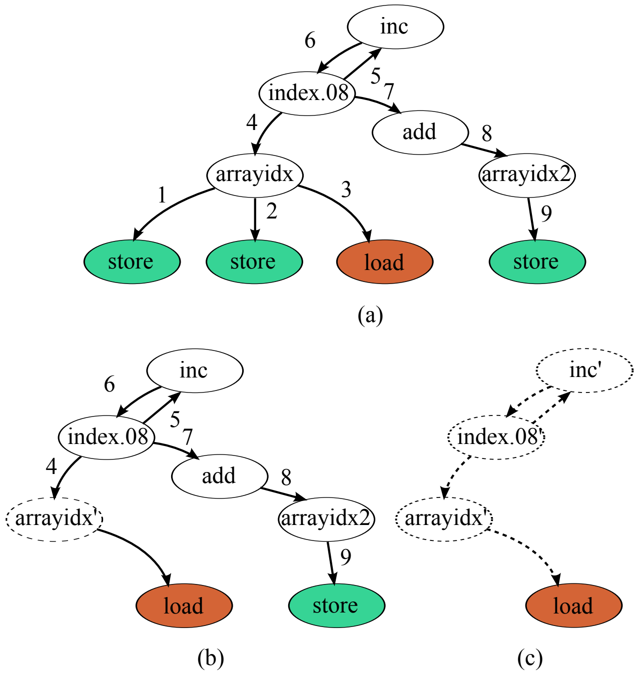

Figure 5: Steps toward contextualization: (a) The data dependency graph (b) The cloned node *arrayidx* eliminates the data dependency between the first two stores and the load (c) Another conflicting path has been found and cloned in order to break the dependency between the load and the last store.

exit condition. Another back-end in LLVM has exactly the same support, Hexagon. Contrary to Hexagon's implementation of the hardware loop builder, we have put our loop builder logic on IR side. New intrinsics representing the hardware loop semantics are inserted automatically by our IR pass. While it requires new intrinsics and additional code for instruction selection, it benefits from the ability of LLVM Scalar Evolution pass to detect trip counts.

Listing 4: cloop.c – Simple C loop

```
for (int i=0; i<200; i++)
  writef(readf());
```

Listing 5: IR output of cloop.c

```
for.body:
  %i.01 = phi i16 [ 0, %entry ],
     [ %inc, %for.body ]
   ...
  %inc = add nsw i16 %i.01, 1
  %exitcond = icmp eq i16 %inc, 200
   br i1 %exitcond, label %for.end,
     label %for.body

for.end:
  ret void
```

Listings 4 and 5 show a simple C loop and its IR output. The transformed code is listed in 6.

The intrinsics *llvm.meph.enter.loop* and *llvm.meph.end.loop* are inserted. This transformation will replace induction variable uses that might become dead after replacement, such as *%inc* and *%i.01* in the transformed code below. We run dead code elimination to clean up the output.

Listing 6: Hardware loop construction

```
entry:
  ; Intrinsics
  call void @llvm.meph.enter.loop(i16 199)
  br label %for.body

for.body:
  %i.01 = phi i16 [ 0, %entry ],
     [ %inc, %for.body ]
  ; Dead IV
  %inc = add nsw i16 %i.01, 1
  ; Dead comparison
  %exitcond = icmp eq i16 %inc, 200
  ; Intrinsics
  %1 = call i16 @llvm.meph.end.loop()
  %2 = icmp eq i16 %1, 0
  br i1 %2, label %for.end,
     label %for.body
```

5.3 Clusterization

The cluster assignation of instructions is a well known problem and many solutions has been proposed in the literature [6, 19, 5]. Starting from the DAG of data-dependent instructions, they try to find a k-way partitioning solution of the DAG based on multiple criteria or cost functions. A lot of effort has been put to the cluster assignation in order to reduce inter-cluster copies, register pressure, balance cluster load, among other parameters, while looking for better overall schedule latency. The problem is addressed either before, during or after instruction scheduling. Unfortunately, Mephisto does not fill the hardware functionalities expected by all these solutions.[3]Even if it provides replicated operators, we have seen that they cannot be arbitrarily associated to instructions. Nevertheless, clusterization still applies on accumulators but under more constrained conditions than previous research work. In section 2.2, accumulator related constraints have been introduced. For the sake of simplicity, we re-enumerate them below:

1. Copies between accumulators are not allowed.
2. Accumulators cannot be spilled. They serve strictly to hold intermediate results.
3. Accumulation is performed on a single accumulator, i.e. two different registers cannot be specified as source and destination of an accumulation operation.

While (2) is addressed by memory promotion passes,[4] (1) and (3) remain open problems and will be handled here.

Mephisto contains two MAC's, MAC0 and MAC1, with two accumulators each one. We have named acc0-1 and acc2-3 the accumulators belonging to MAC0 and MAC1, respectively. Accumulator groups, that we simply called MAC0 and MAC1, are considered as register files or clusters. Widely known clusterization algorithms could have been used here if copies would have been fully, or at least partially, supported. Because register values cannot be transferred,

[3] Homogeneous usage of operators and support for inter-cluster copies.

[4] On the other hand, register allocation might try to add spill code if it runs out of accumulators. Rematerialization has been forced to address this issue

we need to resort to different algorithms for clusterization in order to avoid inter and intra-cluster copies.

Our clusterization pass builds the data dependency graph called *AccDef* tree. Nodes of this tree represent virtual accumulator definitions, and edges definition-use relationships. Property AccDef tree does not have join points other than those introduced by ϕ nodes.

This result follows from the fact that there is no instruction in Mephisto ISA that defines an accumulator register while taking two or more accumulators as source operands.

We deduce from the above property that leafs of AccDef cannot have more than one predecessor.

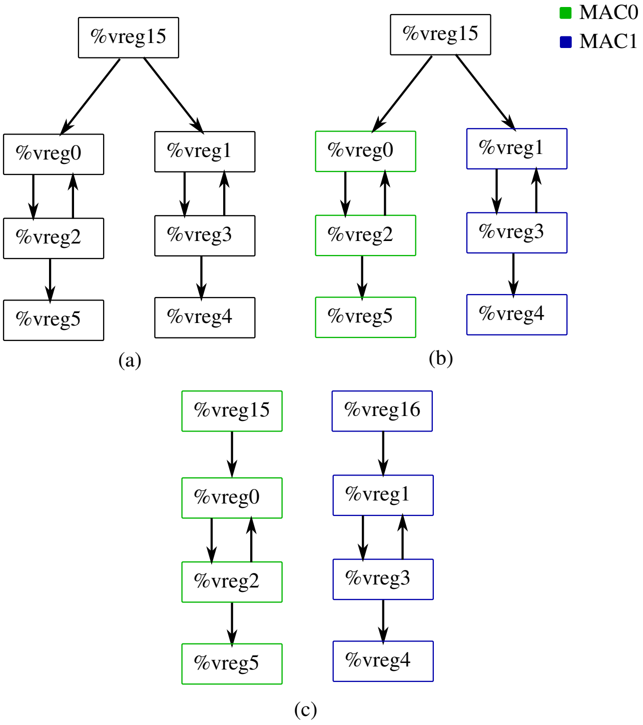

Figure 6: Clusterization steps: (a) The AccDef tree (b) Preclusterization (c) Cloning based on previous phase to avoid conflictive nodes.

We have divided the accumulator clusterizer into two phases: *Pre-Clusterization* and *Clusterization and Cloning*. The "Pre-Clusterization" phase starts by assigning a cluster to each tail or leaf node. Then, it traverses the tree in a depth-first manner looking for dominance properties between the current node and tails. According to this information, the node is assigned to the cluster to which the dominated tail belongs. If it cannot decide, it is left unassigned and conflicts are resolved later by the next phase. Strong connected components in the tree are taken into account and kept in the same cluster.

The "Clusterization and Cloning" phase consists in *node clustering or duplication* if *conflicting edges* are found. Conflicting edges are those that link either non-clustered and clustered nodes or simply nodes with different affected clusters. The former case is handled by cluster assignation if possible while the latter will duplicate nodes to break data dependencies.

Figure 6 shows the AccDef tree of an existing algorithm. We can see that virtual accumulator *%vreg15* is duplicated because it cannot decide whether *%vreg15* should be assigned to MAC0 or to MAC1. It is worth to note that cloning does not always imply a performance penalty in Mephisto as newly inserted nodes can be hidden by the packetizer.

We have designed the clusterizer to avoid inter-cluster copies but intra-cluster ones have not been treated yet. We strongly rely on the register coalescer for intra-cluster copy elimination.

5.4 Register allocation

Register allocation in LLVM is performed after SSA destruction, like most production compilers, e.g. gcc.[5] It is currently far from the effectiveness of register allocations in older compilers, but very flexible and relatively easy to retarget. Figure 7 show the standard pipeline of passes for register allocation.

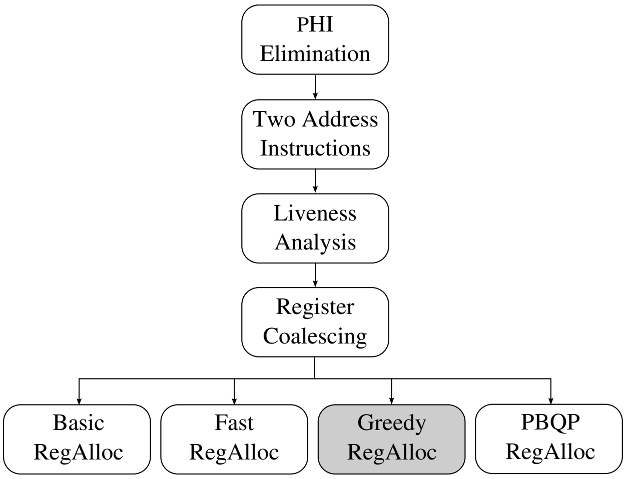

Figure 7: Standard register allocation pipeline in LLVM.

The main register allocator is the Greedy one. Additionally, LLVM provides several allocators for research purposes. The Greedy allocator is based on the linear scan technique detailed in [21] with advanced heuristics for live interval splitting and spilling [18]. Because of the implemented backend model, register allocation in Mephisto becomes somewhat special.

The data register file organization cannot be compared to other widely known VLIW architectures, such as TriMedia [24] or TI C6000 [10], where the output ports are not explicitly included in the ISA. A similar register file organization is proposed in [26] for which no compilation-related work has been found. Hence, we propose a multi-phase allocation scheme, described in figure 8, to efficiently handle both allocatable spaces: the data register file and its output register ports. The approach is simple, an allocation level is associated to register classes and registers are allocated in-order following the register class they belong to.

We expose data register file slots as usual static registers in order to let the allocator make an efficient use of this space. Actually, the allocated registers are memory slots,[6] and loads/stores should be used to read/write its values.

[5] http://gcc.gnu.org

[6] From a hardware point of view, it is designed as an array

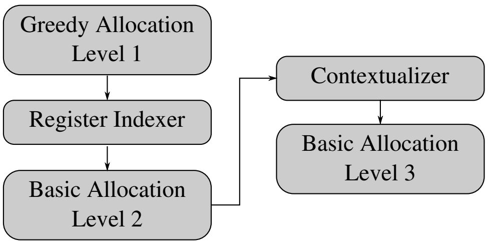

Figure 8: Multi-phase register allocation for Mephisto.

Then, the greedy allocation output requires additional transformations.

The *Register Indexer* pass will come after to insert loads and stores according to the access required at each instruction.[7] It will also add precise alias information in order to avoid unnecessary memory dependencies improving the scheduled code. The output ports are then allocated by the basic allocator at level two.

Spill code. The level two basic allocator selects one register from the available list or, if it is empty, adds spill code to remove the interference. Listing 7 shows an example where two loads are inserted by the register indexer and two new virtual registers are created in consequence. These virtual registers represents the output ports and will be allocated at level two.

Listing 7: Register indexer: add loads to virtual output registers

```
acc += mul r1 , r2
// *** Translated to *** //
load %virt_out0 , [$1]
load %virt_out1 , [$2]
acc += mul %virt_out0 , %virt_out1
```

Mephisto provides only two output ports and spill code plays an important role. Our approach makes spill code of output ports straightforward.

Every time the allocator needs to save and reload the current value of certain virtual output register, *load rematerialization* is performed. The loaded value comes from an allocated slot at allocation level one and it is guaranteed to be preserved until its last use. Therefore, the spiller can rematerialize new loads at every virtual output port use if necessary without incurring in memory aliasing problems.

The main drawback of this approach is the lack of indirect register addressing, which is one of the key features of Mephisto. In order to tackle this issue, we have allowed a dynamic management of the register file.

Dynamic allocated registers. The data register file is split in two regions: static and dynamic. The static region is statically managed by the greedy allocator as detailed above, while the dynamic one is user-defined and represented as an array of integer vectors from high level languages. It exposes virtual output ports earlier in the compilation pipeline but its allocation is delayed till after the register indexer pass.

Dynamic regions enable full use of address generators capabilities. As an indirect consequence, the scheduler will be able to remove false positive memory dependencies based on the fact that both regions do not overlap.

The contextualizer. Each output port has distinct associated register pointer files (Pra and Prb) with its corresponding arithmetic units. Therefore, the bank of register pointers to be used will be fixed only after the second allocation level. For this reason, virtual registers intended to address the dynamic region are left to the last allocator.

Following the selected output register, the *Contextualizer* splits address calculations to fit into the right arithmetic unit along with its register pointer file. Its behavior is equivalent to the contextualizer already described in section 5.1.

6. PRELIMINARY RESULTS

The work presented in this paper is mainly focused on the correctness of generated code. The full set of constraints introduced by Mephisto has been successfully modeled in LLVM allowing us to go further in the compilation process. Currently, a custom assembler verifies code correctness and produces binary code. The preliminary results come from an evaluation of our back-end from a code size perspective. Functional simulations have been performed successfully on a subset of algorithms. Unfortunately, the functional simulator does not allow us to make performance quantifications regarding execution time. In order to fully understand the results, we will give a brief introduction to the binary organization in Mephisto. For further details, we strongly recommend to read [3].

Mephisto implements a two-level instruction cache organization. The instruction is 64 bit wide and split into indexes, which are used to address the second level cache. At the second level, there is a total of 8 tables called *profile tables.* The set of profile table entries specified by the indexes of the instruction at the first level cache forms the VLIW instruction, which is 270 bit wide. Therefore, the actual desired behavior of operators is specified in this cache. Figure 9 shows the cache relationship. The last field of the instruction is a sequence command for control flow specification. Table 1 details the controlled operators per profile table. This organization enables a great level of code compression by reusing table profiles entries.

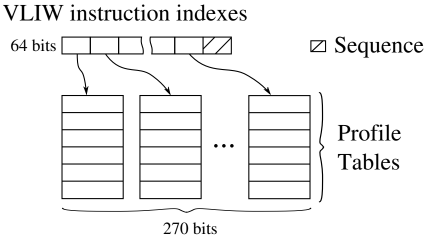

Figure 9: Instruction cache architecture

We use two algorithms for the evaluation, which are a part of the LTE implementation: MIMO encoding and Carrier Frequency Offset (CFO) Tracking. Table 2 shows the size of profile tables and first level cache between the code automatically generated by LLVM and its handwritten version (implemented using the framework proposed in [3]).

of registers with indirect addressing capabilities via register pointers as introduced earlier.

[7] Redundant loads are eliminated.

Table 1: Profile table and operators

Profile Table	Operators
MACt	MAC0 & MAC1
AGmat	Address Generator of memory A
AGmbt	Address Generator of memory B
AGat	Address Generator of register file Port a
AGbt	Address Generator of register file Port b
AGwt	Address Generator of register file for write accesses
Auxt	CORDICs, Divider, Comparison
FIFOt	Input and output FIFOs

Table 2: Size of profile tables and first level cache

	Code generated by LLVM		Handwritten code	
	MIMO encoding	CFO	MIMO encoding	CFO
MACt	8	16	3	9
AGmat	1	6	0	0
AGmbt	0	1	0	0
AGat	3	3	4	8
AGbt	1	2	0	5
AGwt	8	15	2	6
Auxt	0	4	0	4
FIFOt	2	2	1	3
Size of first level VLIW cache				
	32	75	10	26

First of all, we notice that certain address generators are not used at all in the second case. This is due to the fact that both cases have not been implemented *exactly* in the same way. For instance, we have used local variables instead of using data register file mapped arrays. Local variables are stored in memory A and hence increments its corresponding profile table size.

The generated code is ISA compliant but not fully optimized. Many architecture specific optimizations are possible in Mephisto. For example, it implements the *delayed issue* technique introduced in [9]. Given its cache organization, this technique allows a high degree of code compaction, which will reduce the usage of first level cache. In [3], a lot of hardware particularities must be taken into account when coding, including exposed instruction latencies, valid instruction bundles, *etc.* Low level coding is an error-prone and time consuming task. LLVM offers a good trade-off between programmability and efficiency but we still need to work on custom optimizations passes in order to improve the code performance.

7. RELATED AND FUTURE WORK

Code generation for embedded processors, micro-controllers and ASIPs is a mature domain of research and innovation, with branches and challenges in many technical areas [17, 12].

Among these, code generation for clustered VLIW processors is a difficult problem and has been an active research area until recently, starting with Ellis and the bottom-up greedy (BUG) algorithm [6]. The unified cluster assignment (UAS) [19] is also a popular heuristic. Eriksson et al.[7] propose a comprehensive survey of the combined scheduling and cluster assignment problem, as well as an optimal (albeit non scalable) algorithm. On the other hand, post register allocation scheduling with accurate information about instruction latencies is necessary when targeting non-interlocked processors such as Mephisto. We have implemented a custom scheduler with additional heuristics complementing the current VLIW-aware LLVM scheduler. However, we lack of interesting scheduling techniques, e.g. software pipelining [2], which are quite profitable considering the underlying architecture and our application domain.

Induction variable detection has been studied extensively because of its central role in loop optimizations [25]. LLVM implements a variation of the technique proposed by Pop [22]. In this paper, it is primarily useful to support induction variable canonicalization [16], and to generate code for address generators [8, 15].

SDR algorithms are difficult to implement efficiently in classical languages such as C, even with its DSP extensions [11]. Usually these integer-based algorithms result from a discretization of a floating point-based algorithm. But the C language, even with DSP extensions, makes it difficult to reflect and take into account all the peculiarities, constraints and opportunities that come with the discretization decisions. These decisions are highly correlated with middle-end optimizations such as partial redundancy elimination and vectorization, and back-end passes such as instruction selection, scheduling, and register allocation. The co-design of data types and operators (possibly in a domain-specific language) is one promising direction, with some interesting advances in high-level synthesis [27]. A more comprehensive study of the optimization space and operations research modeling should also be conducted.

8. CONCLUSION

Application specific VLIW processors are well suited to the implementation of complex algorithms within highly constrained environments. Application specialization and power/energy optimizations tend to go against programmability and to raise the barrier for the automatic generation of efficient code.

We presented some important architectural constraints of Mephisto, a highly specialized and power-optimized VLIW processor. We also presented our implementation of an automatic code generator based on LLVM that overcomes these constraints. Among these, we isolated the addressable register files as one of the most interesting scientific and engineering challenges. We introduced *contextualization*, an algorithm to partition the dependent computing of data and addresses, going beyond the decoupling induced by traditional address generators. We also proposed a specific back-end compilation flow, decomposing register allocation into three-steps. Many other constraints not included in this paper—but not less important—have handled, to allow the generation of ISA compliant code. Our preliminary evaluation is encouraging, but further work on code optimization and on language extensions is required to improve the performance of the generated code.

Acknowledgments. This research is partially supported by the European Commission under the 7th Framework program within the ToucHMore project (FP7 ICT-288166) [1].

9. REFERENCES

[1] Automatic Customizable Tool-chain for Heterogeneous Multicore Platform Software Development (ToucHMore).

[2] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software Pipelining. *ACM Comput. Surv.*, 27(3):367–432, Sept. 1995.

[3] C. Bernard and F. Clermidy. A low-power VLIW Processor for 3GPP-LTE Complex Numbers Processing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[4] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based Digital Baseband for MIMO 4G SDR. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 278–279, Feb. 2010.

[5] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. *HP Laboratories Technical Report HPL*, Jan 1998.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, New Haven, CT, USA, 1985. AAI8600982.

[7] M. V. Eriksson, O. Skoog, and C. W. Kessler. Optimal vs. heuristic integrated code generation for clustered vliw architectures. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, SCOPES '08, pages 11–20, New York, NY, USA, 2008. ACM.

[8] D. Grant and P. B. Denyer. Address generation for array access based on modulus m counters. In *European Design Automation Conference*, 1991.

[9] J. P. Grossman. Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation.

[10] T. I. Inc. TMS320C6000 Technical Brief. Technical report, Texas Instruments Inc., February 1999.

[11] ISO/IEC. Programming Languages - C - Extensions to Support Embedded Processors. Technical report, April 2006.

[12] M. K. Jain, M. Balakrishnan, and A. Kumar. Asip design methodologies: Survey and issues. In *Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*, VLSID '01, pages 76–, Washington, DC, 2001. IEEE Computer Society.

[13] S. Larin and A. Trick. BOF: Instruction Scheduling for Superscalar and VLIW Platforms. Temporal Perspective. LLVM Developers' Meeting, 2012.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, California, Mar 2004.

[15] C. Liem, P. Paulin, and A. Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *Design and Automation Conference (DAC '96)*, 1996.

[16] S.-M. Liu, R. Lo, , and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *In Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT'96)*. IEEE Computer Society, 1996.

[17] P. Marwedel and G. Goosens, editors. *Code Generation for Embedded Processors*. Kluwer, 1995.

[18] J. Olesen. Register Allocation in LLVM 3.0. LLVM Developers' Meeting, 2011.

[19] E. Özer, S. Banerjia, and T. M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 308–315, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[20] J. Park, S.-B. Park, J. D. Balfour, D. Black-Schaffer, C. Kozyrakis, and W. J. Dally. Register Pointer Architecture for Efficient Embedded Processors. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 600–605, San Jose, CA, USA, 2007. EDA Consortium.

[21] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 21(5):895–913, 1999.

[22] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *(HiPEAC)*, number 3793, pages 218–232, Barcelona, Spain, Nov. 2005.

[23] M. Postiff and T. Mudge. Smart Register Files for High-Performance Microprocessors. Technical report, 1999.

[24] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen. The TM3270 Media-Processor. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12 pp. –342, Nov. 2005.

[25] M. J. Wolfe. Beyond induction variables. In *(PLDI'92)*, pages 162–174, San Francisco, CA, June 1992.

[26] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level Hierarchical Register File Organization for VLIW Processors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 137–146, New York, NY, USA, 2000. ACM.

[27] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level optimization for high-level synthesis and fpga-based acceleration. In *FPGA '10*, Monterey, California, Feb. 2010.