# Memory Hierarchy Hardware-Software Co-design in Embedded Systems

Zhiguo Ge[1], H. B. Lim[2], W. F. Wong[1,2]
[1] Department of Computer Science, [2] Singapore-MIT Alliance, National University of Singapore

***Abstract*— The memory hierarchy is the main bottleneck in modern computer systems as the gap between the speed of the processor and the memory continues to grow larger. The situation in embedded systems is even worse. The memory hierarchy consumes a large amount of chip area and energy, which are precious resources in embedded systems. Moreover, embedded systems have multiple design objectives such as performance, energy consumption, and area, etc.**

**Customizing the memory hierarchy for specific applications is a very important way to take full advantage of limited resources to maximize the performance. However, the traditional custom memory hierarchy design methodologies are phase-ordered. They separate the application optimization from the memory hierarchy architecture design, which tend to result in local-optimal solutions. In traditional Hardware-Software co-design methodologies, much of the work has focused on utilizing reconfigurable logic to partition the computation. However, utilizing reconfigurable logic to perform the memory hierarchy design is seldom addressed.**

**In this paper, we propose a new framework for designing memory hierarchy for embedded systems. The framework will take advantage of the flexible reconfigurable logic to customize the memory hierarchy for specific applications. It combines the application optimization and memory hierarchy design together to obtain a global-optimal solution. Using the framework, we performed a case study to design a new software-controlled instruction memory that showed promising potential.**

***Index Terms*— Memory hierarchy design, embedded systems, reconfigurable logic.**

## I. Introduction

Embedded systems have different characteristics compared to general-purpose computer systems. First, they combine software and hardware to run specific applications that range from multimedia consumer devices to industry control system. These applications differ greatly in their characteristics.They demand different hardware architectures to maximize performance and minimize cost, or make a tradeoff between performance and cost according to the expected objectives. Second, unlike general-purpose systems, embedded systems are characterized by restrictive resources and low energy budget. In addition to the vigorous restrictions, embedded systems have to provide high computation capability and meet real-time constraints.

Zhiguo Ge is with the Department of Computer Science, National University of Singapore. Email:{gezhiguo@comp.nus.edu.sg}

H.B. Lim is with the Singapore-MIT Alliance, National University of Singapore. Email:{limhb@comp.nus.edu.sg}

W.F. Wong is with the Department of Computer Science and the Singapore-MIT Alliance, National University of Singapore. Email:{wongwf@comp.nus.edu.sg}

All these diverse constraints on embedded systems including area, performance and power consumption result in enormous issues and concerns during the design process. Among them, memory hierarchy design is of great importance. The memory bottleneck in a modern computer system is a widely known problem: the memory speed cannot keep up with the processor speed. This problem becomes even worse in an embedded system, where designers not only need to consider the performance, but also the energy consumption. In an embedded system, memory hierarchy takes a huge portion of both the chip area and power consumption. Thus, optimizing the memory hierarchy to reduce hardware usage and energy consumption in order to sustain high performance becomes extremely important.

Basically, we categorize the optimization methods for memory hierarchy into two approaches. The first approach deals with the architectural aspect, where designers customize and tune the memory hierarchy by analyzing specific applications, including parameterizing the data cache size and line size, instruction cache size, scratch memory size, etc. The second approach deals with the software aspect, where designers analyze and optimize the application intensively, such as partitioning data into different types of storage, optimizing the data layout to reduce the amount of cache conflicts, etc. Most previous research focus on the two methods separately. Researchers either perform application optimizations for a given memory architecture, or design a memory architecture by analyzing applications. However, these two approaches may affect each other.

In order to explore the design space more thoroughly and make the hardware and software match better, it is necessary to combine these two aspects. Borrowing the idea and concept from Software and Hardware Co-design, we propose a new Memory Hierarchy Co-design methodology to design embedded system memory hierarchy. In this framework, the management of the application and architecture will be done uniformly, and they will interact with and guide each other. The framework takes the application source code, hardware information, and objectives constraints as inputs. It outputs the transformed source code and the memory hierarchy architecture specifically for the transformed source code.

The rest of this paper is organized as follows. In Section 2, we present the background and motivation for Memory Hierarchy Co-design. Section 3 discusses the memory hierarchy architecture parameterization. We explain the key software techniques for program analysis and transformations that our framework require in Section 4. In Section 5, we present

a preliminary case study of the feasibility of our approach. Finally, we conclude in Section 6 and discuss our ideas for future research.

## II. Background and Motivation

### A. Embedded System-on-Chip Design and Memory Hierarchy Issues

*1) Embedded system-on-chip:* As modern digital systems become increasingly more complex and time-to-market becomes shorter, the task of designing digital systems becomes more and more challenging. New design methodologies have to be developed to cope with this problem, such as component reuse of processor core, memories and co-processor etc. System-on-Chip (SOC) refers to products that integrate the components into one system on a single silicon chip.

One prominent characteristic of the embedded SOC is its heterogenous architecture with programmable processor cores, custom designed logic, and different type of memory on a single chip [1]. The architecture can be tailored or reconfigured for specific applications. Custom instructions intended for specific applications can dramatically improve the CPU performance. Tuning the cache size, associativity, and line size can best make use of the limited hardware resources to achieve maximal performance for the application. For most cases, customizing the hardware architecture requires sophisticated compiler support for generating the corresponding code for the customized architecture. Thus, the compiler and hardware both need to be managed together in the design of embedded systems. In the Software-Hardware Co-design methodology, the design process starts with one uniform description for both hardware and software. The computations are partitioned into different functional units such as processor, co-processor and hardware functional units, etc.

The nature of the heterogeneous architecture and the tightly-coupled hardware and software of embedded systems result in many research issues involving both architecture and software co-optimization. The software optimizations and hardware optimizations should be accomplished in one uniform framework, which is seldom addressed in traditional research on embedded systems.

*2) Embedded memory hierarchy issues:* The memory is a bottleneck in a computer system since the memory speed cannot keep up with the processor speed, and the gap is becoming larger and larger. Memory hierarchy issues are among the most important concerns in designing application-driven embedded systems. The memory system occupies a large area and consumes a lot of energy. Studies show that in many digital signal processing applications, the memory subsystem occupies up to ten times the area and power consumed than those of the data-path [1].

Figure 1 shows a typical embedded system architecture consisting of processor core, reconfigurable hardware, instruction cache, data cache, on-chip scratch memory, on-chip DRAM, and off-chip memory. As shown in the figure, the computations are partitioned into different computational units while the data are assigned to different storage components.

Unlike the memory hierarchy of a general-purpose system mainly concerned with performance, the memory hierarchy of an embedded system has more diverse design objectives, including area, performance, and power consumption.

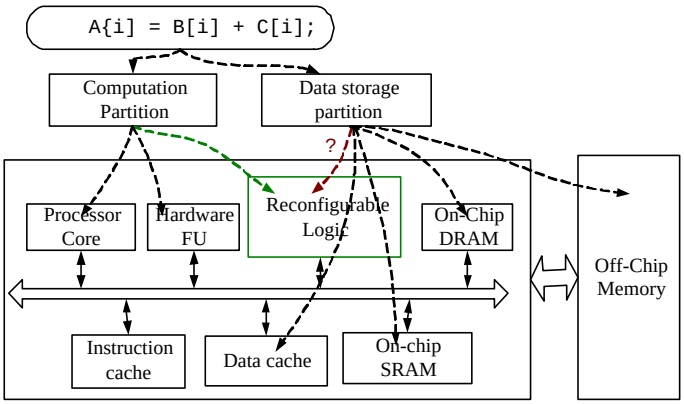


Fig. 1. A typical embedded system architecture

On one hand, researchers tune the memory hierarchy for specific applications [2]. On the other hand, they optimize the applications to reduce the memory size needed [3], [4], and optimally assign the data into storage [5] to maximize the performance and decrease the energy consumption by reducing the amount of data transfer between the processor and off-chip memory [6]. Recently, researchers are beginning to focus on combining these two aspects. The application is optimized and transformed, meanwhile the architecture is reconfigured accordingly [7].

### B. Traditional Memory Hierarchy Design Methodology

Traditionally, system designers either perform software optimization independently and assign the data to the allocated memory units for the optimized applicatione code [8], or design the most suitable memory hierarchy for a given application [9]. They seldom consider the influence between the architecture exploration and the application optimization.

*1) DSTE framework:* The Data Transfer and Storage Exploration (DTSE) methodology, developed by the Interuniversity Microelectronics Center (IMEC) within the context of the $ATOMIUM$ research project [8], is well known for managing memory architecture and embedded applications, especially multimedia applications.

Many embedded applications are data-dominated and experiments show that data transfer and data storage consume a large amount of energy and chip area. ”The goal of DSTE is to determine an optimal execution order for the data transfer together with an optimal memory hierarchy for storing the data of given application” [8]. By transforming the application source code to reduce the need for storage and the need for data transfer between CPU and off-chip memory, the energy consumption and the required storage size are decreased dramatically. Figure 2 shows the design flow of the DTSE methodology.

From the figure, the methodology is basically top-down. Starting from the system level specification, it performs optimization step by step from memory architecture independent optimization down to the implementation of the system. There

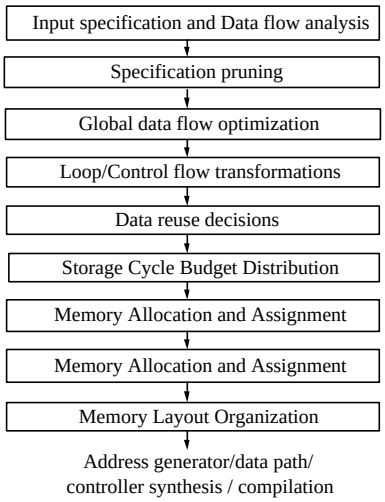


Fig. 2. Design flow of DSTE methodology [8]

are no iterations between different phases and the loop/control flow transformations are completely architecturally independent. In general, the higher level optimization should produce larger performance gain.

*2) MemExpore framework:* Instead of optimizing application, Panda et. al. [1] presents a framework, named MemExplore, for designing the on-chip memory hierarchy for a specific application based on the analysis of the application. Given a certain amount of on-chip memory, the problem is to partition it into on-chip scratch memory and data cache to maximize the performance. The cache is characterized by its size (C) and line size (L), and the scratch memory is characterized by its size. The metric used for evaluating the memory architecture is the total number of processor cycles required for the memory access. The optimal architecture should minimize this metric. Apart from determining the memory architecture , the application data is assigned to the memory storage units such as on-chip scratch memory and off-chip memory (accessed through cache).

For an on-chip memory size T, the algorithm divides it into varying amount of cache (size C and line size L) and scratch-pad memory. The framework partitions and assigns the most critical scalars and arrays into scratch-pad memory based on the data size, memory access frequency, and the degree of cache conflict. Analytical models are used to estimate the performance. The pair (C,L) which is estimated to maximize performance will be selected.

### C. Drawback of the Traditional Methodologies

Most of the existing research on memory hierarchy either focuses on application optimization without considering the memory architecture, or explore suitable memory architectures for given applications.

In fact, that the architecture and the application optimizations affect each other. The current methodologies are phase-ordered and thus separate the two tasks, (i,e, application optimization and memory architecture design).

### D. The Framework of Memory Hierarchy Co-design

The traditional methods have limited design instance space available to search, which will probably result in a suboptimal solution. To avoid getting a suboptimal solution and to obtain a global optimal one, we propose a new framework in this paper.

*1) Rationale of framework:* The rationale of the proposed framework is to search for an optimal solution from a much larger design instance space which includes all the combinations of application optimization and memory hierarchy architecture. Theoretically, the new method is able to obtain the global optimal solution in terms of application optimization and memory hierarchy architecture.

*2) Framework design:* Our framework for memory hierarchy co-design is shown in Figure 3. This framework takes the application, hardware information, and objectives as inputs. It will exploit software transformations and optimizations, explore memory hierarchy architectures, and evaluate the performance and other objectives.

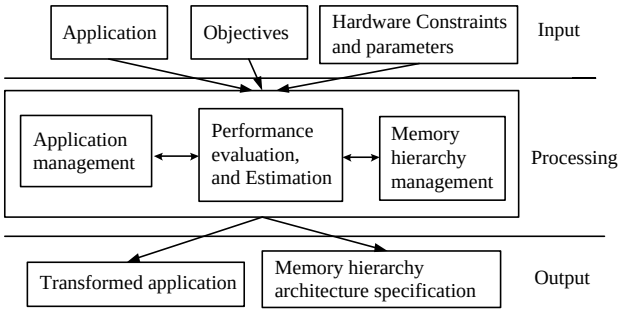


Fig. 3. The framework of memory hierarchy co-design

The input application can be system-level source code, intermediate representations, or binary code. The hardware input can be the information and constraints of the hardware resources for memory hierarchy, such as the amount of hardware resources available, how much on-chip DRAM and SRAM, and the amount of reconfigurable resources, etc. The objectives are the metrics we need to focus on, such as performance, energy consumption and real-time constraints, etc. Different applications or devices may require different objective metrics. For example, energy consumption is of great importance to handset devices which have very limited energy budget, while real-time property is very crucial for control applications and devices which must guarantee the response and service within certain time deadline.

The processing portion of the framework consists of three parts: application management, memory hierarchy management, and performance evaluation and estimation. The application management part is responsible for performing application and loop transformations and optimizations. The memory hardware management part is to explore the suitable memory hierarchy architecture for optimized applications. The performance evaluation and estimation part is used to direct the application optimization and the exploration of memory hierarchy.

The output of the framework are the transformed and optimized application code, and the corresponding memory

hierarchy architecture specifications. Specifically, the specifications of memory hierarchy architectures can be synthesized and implemented on hardware platforms such as ASIC and FPGA.

The key difference between our proposed framework and the traditional methods is that the application management and memory hardware architecture management direct each other. Unlike the DSTE method which performs loop transformation independently, the loop transformation and application optimization of the proposed framework is based on the memory hierarchy management, while the management of memory hierarchy is influenced by the application optimizations.

## III. Memory Hierarchy Architecture Parameterization

### A. Tuning Memory Hierarchy Architecture for Applications

Studies show that applications from different domains exhibit different characteristics [2]. In a general-purpose system, a large cache is often used to fit all the applications, which results in resource consumption of up to 80% of the total transistor budget and up to 50% of the die area [2].

For resource stringent embedded systems, it is not a good strategy to meet the performance requirements with large caches. The huge energy consumption by large caches is not tolerable for embedded systems. In order to decrease energy consumption, some cache architecture variations have been developed according to the principle that accessing smaller storage units consumes less energy.

For example, a phased-lookup cache uses two-phase lookup. All the tag arrays are accessed in the first phase, then only the data way corresponding to the hit tag way will be accessed [10]. At the expense of larger access latencies, this strategy results in less power consumption since a smaller data way will be accessed.

Way predictive set-associative caches access only one tag and data array initially. If a miss happens, then the other ways will be accessed [10]. Filter caches are very small direct-mapped caches in front of the regular cache [10]. If most accesses fall into filter caches, power will be saved since accessing the small filter caches consume less power.

In addition to the above techniques, the application-driven property of embedded system offers a big opportunity for tuning the memory hierarchy specifically for applications. Statically parameterizing memory hierarchies according to the demands of the applications has been studied quite thoroughly in previous research. In the DSTE methodology [8], different memory units with different number of ports and different word widths are allocated according to application analysis and instructions scheduling. MemExplore [1] explores the best memory hierarchy variants consisting of cache and scratch memory for the given applications.

Recently, researchers have started to focus on configurable cache architectures. The cache architectures can be changed by the control of applications. A program is able to configure the memory architecture when it starts execution or during the course of its execution according to its needs. [10] presents a reconfigurable cache architecture for a general-purpose processor, where the cache banks can be reconfigured as instruction reuse buffer.

Basically, the parameterization of memory hierarchy can be classified into two categories: static parameterization (for statically configurable memory hierarchy) and dynamic parameterization (for dynamically reconfigurable memory hierarchy).

### B. Potential Opportunities for Memory Hierarchy Parameterization

The possible opportunities for parameterizing the memory hierarchy are as follows:

**Utilizing the limited resources according to demand.** By parameterizing the memory hierarchy according to the requirement of applications, high performance can be sustained with minimal resource usage. Different applications require different memory hierarchy architectures. Some applications may need large caches to obtain high performance, while for other applications, small caches may be enough to achieve high performance. Based on different requirements from applications, allocating memory units on demand can save much resource and obtain high performance.

**Handling program execution phases.** Apart from the different requirements for memory hierarchies from different applications, different program execution phases may require different memory hierarchies. Thus, dynamically configuring the memory hierarchy on the fly to meet the needs of different execution phases of a program becomes potentially beneficial.

**Real-time requirement.** Real-time property is very important for embedded applications. In certain cycles budget, the feasible instruction scheduling may drive the need for multiple simultaneous memory accesses [11]. Thus, elaborately designed memory hierarchy is indispensable for these applications. On the other hand, many DSP processors use scratch memories instead of caches to ensure the predictability of memory latencies [2].

**Need of compiler or software controlled memory [2].** Caches make use of hardware mechanisms for replacing data. The hardware replacement policy is not flexible, and thus it may result in many cache conflicts. In fact, certain data replacements are predictable. Thus it is desirable for the compiler to determine such data replacements to avoid the cache conflicts. The scratch memory is an important software-controlled memory variant [12], [13].

### C. Static Memory Hierarchy Parameterization

In static memory hierarchy parameterization, the memory architecture is determined statically. The applications cannot change the configuration of the memory hierarchy. Basically, the memory hierarchy is parameterized by analyzing the application. Once the memory hierarchy is implemented in hardware, such as ASIC or FPGA, the application cannot change the architecture before and during the execution. The parameters that can be configured are the number of memory banks, the bit widths of the memory units, the port numbers of the memory units, the cache size, the cache line size, the cache associativity, the on-chip memory size, and the number

of registers, etc. The static memory hierarchy parameterization normally determines a memory hierarchy for a specific application, because the application cannot change the memory hierarchy once it has been implemented.

A method to parameterize the memory hierarchy for real-time applications is presented in [11]. If two load operations to the same memory banks must be scheduled at the same cycle to meet the real-time requirement, the memory banks must be parameterized to have dual read ports.

A memory exploration framework is presented in [1]. Given a fixed size memory budget, it partitions the memory into some amount of cache and scratch memory to avoid the conflicts between data accesses. The most frequently used data with small size will be put into the scratch memory to avoid the conflicts.

### D. Dynamic Memory Hierarchy Parameterization

With dynamic memory hierarchy parameterization, the memory hierarchy can be changed during the course of program execution. Thus, the dynamic memory hierarchy parameterization offers applications the flexibility to configure the memory hierarchy according to its need. There are two ways to exploit this technique. First, dynamic memory hierarchy parameterization makes it possible to customize memory hierarchies for a variety of applications rather than just one application. Since the application can change the memory hierarchy, various applications can tune the memory hierarchy for their own needs. Second, it provides the chance for an application to change the memory hierarchy during its different execution phases. Different execution phases may have different requirements for the memory hierarchy. Dynamic memory hierarchy parameterization provides the application with some flexibility to reconfigure the memory hierarchy to meet the requirements for different phases.

A reconfigurable cache architecture for general-purpose processors is presented in [2]. The L1 data cache is partitioned into two banks. One of them is used as conventional cache, while the other can be configured as an instruction reuse buffer [2]. For some multimedia applications, the data streaming characteristics cause large caches to be under-utilized. In order to make better use of the hardware resources, such applications can configure one of the banks of the under-utilized cache into an instruction reuse buffer at the beginning of application execution.

Using a new technique, called way concatenation [10], the cache associativity can be configured dynamically. The cache associativity greatly affects power consumption. Studies show that a direct-mapped cache consume 30% energy of a same size four-way set associative cache [10]. The reason for the low power consumption of the direct-mapped caches is that only one tag and one data array are read during an access, while four tags and four data arrays are read for a four-way associative cache. For some applications, a direct-mapped cache has a low miss rate, and thus results in low energy consumption. However, other applications result in a high miss rate for direct-mapped cache. This high miss rate incurs larger energy consumption due to the longer execution time and more energy-consuming accesses to the larger low-level memory units. Thus, using a suitable cache associativity for a particular application is of great importance to decrease energy consumption.

## IV. Program Analysis and Transformations

### A. Data-flow Analysis

*1) Standard data-flow analysis:* Data flow analysis is an important technique to obtain useful information about the flow of data in a program, such as the uses and definitions of data [14]. Most data flow analysis is performed statically at compile time.

There are various representations for data flow analysis. The most commonly used are as flows: the *directed acyclic graph* (DAG), the *control flow graph* (CFG), the *call graph* (CG), and the *static single assignment form* (SSA).

Data flow analysis is widely used to optimize the program data flow. Its typical applications are as follows:

- Available Expressions Analysis is used to identify the previously computed values to eliminate redundant computation.
- Reaching Definition Analysis is used to detect and eliminate redundant definition.
- Live Variables Analysis is used to determine whether a variable will be used after a program point.
- Constant Propagation tries to replace an expression by a constant when compiling.

*2) Memory-oriented data flow analysis:* Several models have been proposed for data flow analysis oriented to array data.

**Stream model:** Stream model is used for data description in Phideo compiler [15]. Multiple-dimensional signals are mapped to time axis by use of a linear function. A schedule determines the offset of the stream on the time axis. The drawback of the stream model is that the dependency between the data is not well specified.

**Polyhedral models:** Polyhedral models are widely used for array synthesis systems. A polyhedral model can be used for partitioning array data into non-overlapped portions. It can also be used for variable counting and memory size estimation [8].

### B. Program Transformations and Optimizations

The application optimizations have a large impact on the application performance. There are different application optimizations for different objectives. Data flow optimizations are used to reduce the redundant memory accesses. Loop transformations can improve the data locality. Memory in-place optimizations can reduce the memory size requirement. If application transformations are not applied, the memory organization will be heavily suboptimal [1].

For embedded systems, program transformations should be performed to maximize or make a suitable tradeoff between multiple design objectives, such as performance, memory footprint, and energy.

*1) Data-flow Transformations:* Data flow analysis and transformations happen at the beginning of the design process, which greatly impact the performance of applications. Proper data flow transformations can produce significant performance gains. Data flow transformations may reduce or remove redundant data accesses. Furthermore, data flow transformations may dramatically reduce the size of the intermediate buffers or even eliminate them completely. For example, signal substitution and propagation can eliminate the copies of the same data which are accessed by different loops or kernels. Optimizing the computation order in associate chains can change the data production and consumption order which can result in much lower intermediate buffer size requirement.

This buffer size reduction decreases the requirement for storage. Furthermore, the reduced buffer can be put into the smaller and faster storage units, which in turn results in better performance and less energy consumption.

*2) Loop transformations:* The main purpose of loop transformations for memory hierarchy design is to improve the access locality, including the temporal locality and spatial locality. Apart from being used for memory organization, loop transformations are also used for other purposes, such as to exploit the inherent parallelism of algorithms.

The commonly used loop transformations are loop interchanging, loop tiling, loop merging, loop unrolling, etc [16]. Loop transformations may substantially change the characteristics of the programs. For example, loop transformations can change the spatial and the temporal locality of memory accesses. By moving a certain loop to the innermost level, the inner loop may carry the reuse, and thus improving the temporal locality. As a result, the variables may be stored in smaller and faster storage units or even registers.

Loop transformations can also be used to discover loop parallelism. For example, to parallelize a nested loop, the loop that carries all the dependence should be interchanged to the outermost level. Then, the rest of the loops should be executed in parallel [16].

*3) Code rewriting to improve data reuse:* Optimizing the data transfer and storage of programs is very important for improving the performance of memory hierarchies. Apart from the loop transformations which aim to improve the performance of programs, other code rewriting techniques can also achieve performance improvement. For example, by explicitly adding copies of subsets of the data in the program source code [17], the program explicitly controls the data transfers and the reuse of storage copies to maximize the data reuse.

*4) Memory in-place optimizations:* Another scenario where source code rewriting may be applied is memory in-place optimization [4], [8], which aims to reuse memory locations as much as possible to reduce the memory size requirements.

The principle to reduce the number of required registers is to put the variables in the same location if their life times are non-overlapped. The same principle applies in reducing the memory size requirements. Basically, there are two ways to reduce the memory requirement. The first one is to assign the elements from the same array (*intra-array*) to the same memory locations. The other one tries to put the elements from different arrays (*inter-array*) to the same memory locations.

*5) Storage Bandwidth Optimization (SBO):* The goal of parameterizing and customizing memory architectures is to determine the suitable memory architectures for given applications. One important factor that affects the cost of the customized memory hierarchy is the memory access pattern of the underlying program. The objective of Storage Bandwidth Optimization (SBO) is to lower the cost of the memory hierarchy, in terms of the number of memory ports and the memory banks, by exploiting the scheduling of memory access patterns [8].

*6) Impact of the transformations on multiple design objectives:* A transformation may have different effects on the design objectives. Some effects may conflict with each other. For example, the memory in-place optimizations can reduce the memory size requirements, but the execution time may be increased due to the extra code for managing the signal address. Since embedded systems have multiple design objectives, it is necessary to make a suitable trade-off between these objectives.

Some loop transformations may need the hardware platform support, while some loop transformations need specific platforms to exploit the potential improvement of the transformations. For example, loop merging can be used to parallelize loops, and to decrease the amount of intermediate buffer needed, which in turn improves the performance of data cache. However, merging loops may increase the number of instructions in the loop kernel, which in turn might result in a larger instruction cache size requirement. Combining the source transformations with the architecture parameterization might lead to larger performance improvement.

## V. Software-controlled instruction memory: A Case Study

Based on our framework, we performed a case study to develop a new instruction memory architecture called the software-controlled instruction memory.

### A. Motivation

As the speed of the processor becomes increasingly faster, the instruction fetch architecture that provides the processor with needed instructions has become more and more important. However, the low access speed of the memory can not keep up with the processor speed, and this is a major bottleneck in modern systems. The most important way to tackle this problem is to use small and fast cache memories between the processor and the main memory. However, even a small cache miss rate can incur a large latency penalty that greatly hinders the performance of the processor. Much effort has been made to decrease the cache miss rate.

Instruction placement optimizations are introduced in [18], [19]. The goal is to improve the instruction locality by repositioning the instructions. It can be performed at different granularity. Repositioning the instruction at procedure level is called *Global Layout Optimization*, while performing the optimization at basic block level inside a procedure is called *Function Layout Optimization*. To get a finer granularity than procedure level, we can split the procedure, which is called *procedure splitting*.

### B. Improving the Instruction Memory Architecture

Caches use hardware mechanisms to implement the data replacement. These mechanisms are not flexible and may result in many conflict misses. The characteristics of the program can often be statically analyzed. Designing a memory architecture to make full use of this static analysis information is desirable. For example, by using software-controlled memory architecture, the compiler may have much more freedom to perform the program optimization, and obtain extra performance improvement.

In [9], the on-chip storage resources are partitioned separately into data cache and scratch memory. The program data are carefully assigned into these two memory units to reduce the conflict miss. We believe that a similar approach can be applied for instruction memory units.

Based on the above principles, we propose an instruction memory architecture consisting of the instruction cache and an on-chip scratch memory. The most frequently executed portions of the program will be placed in the on-chip memory, while less frequently executed parts will be handled by the cache. This new architecture can provide thre main benefits. First, by carefully assigning the most frequently executed parts of the program into scratch memory, the conflict misses will be reduced. Second, the on-chip scratch memory is less costly in terms of both resource and energy consumption. Third, since the behavior of the scratch memory is completely predictable, the real-time property of the system can be strengthened.

### C. The Framework and Design Flow

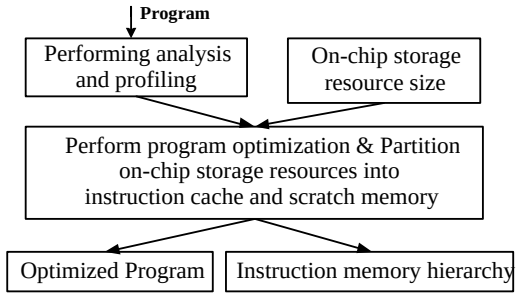


Fig. 4. Design flow

The design flow is shown in Figure 4. The inputs are the on-chip memory storage resource size and the program, either in the form of the source code or intermediate code. Taking the inputs, the framework will explore the software optimizations and the instruction memory hierarchy configurations in order to get the best combination.

The optimizations for the program are to reorder the program blocks and assign the blocks either to the instruction cache or the on-chip scratch memory. The memory architecture is shown in Figure 5. For the memory architecture aspect, the task is to decide the most suitable partition of the on-chip storage resource into the cache and the scratch memory for the program. These two aspects guide each other during the optimization process. Upon completion of the optimization, the framework will output the optimized program and a memory hierarchy architecture for running the optimized program. The memory hierarchy architecture is in the form of a Verilog description which can be synthesized into hardware.

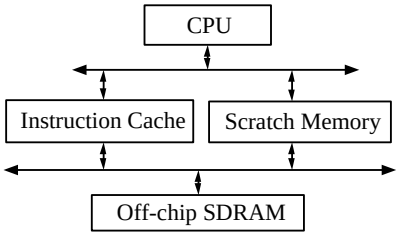


Fig. 5. Proposed memory architecture

In this case study, we will manipulate the program at the assembly code level because we can control the positioning of the instructions, and the processing complexity is acceptable. First, all the source files are compiled into assembly code files. Second, we instrument the assembly code in order to profile the program for obtaining the execution information. Third, we perform the program block positioning according to the profiling information and the hardware resource size. This also enables us to determine the memory architecture parameters. Finally, we compile the optimized assembly code into binary executable and synthesize the memory architecture platform. The executable can then be run on the hardware platform.

### D. Experimental Platform

*1) Hardware platform:* In this case study, we use the Nios Development Kit (Stratix Professional Edition) [20], which is based on the Altera Nios processor and the Stratix EPS10 FPGA, to prototype the system. Nios is a highly reconfigurable soft-core CPU, with customizable instructions and customizable cache size, etc.

The Stratix EPS10 FPGA is connected to an external memory hierarchy consisting of 16M bytes off-chip SDRAM, 1M bytes off-chip SRAM, 8M bytes off-chip flash memory, and 16M bytes compact flash disk.

*2) Software tools:* The software tools provided include the SOPC, the QuartusII, and the Nios processor software development kit. SOPC is a software tool for integrating all the *intellectual property* (IP) components into an on-chip system, and generating the corresponding hardware language specifications of the system. QuartusII is a software tool for synthesizing and implementing the hardware language specifications into hardware for running applications. The Nios processor software development kit is responsible for compiling the programs into their executables for Nios based systems.

### E. Experimental Methodology

*1) Hardware setup:* In this case study, we modified the Verilog source code of the Nios processor by adding our custom memory hierarchy. We have performed the following work:

1. Modified an existing direct-mapped cache design into a two-way set-associative cache. The cache size can be easily changed.
2. Implemented a scratch memory besides the on-chip instruction cache, which can be accessed in parallel with the cache by the Nios processor.
3. Placed the instructions of selected subroutines manually into the scratch memory.
4. Introduced a timer and hardware counters into the system to collect performance statistics such as the program execution time, the number of instructions fetched by the processor, and the instruction cache miss rate.

*2) Benchmark application:* For our experiments, we have selected the FFT (Fast Fourier Transformation) from the MiBench benchmark suite, which is mainly used for evaluating embedded systems. FFT is used to perform fast fourier transformation computation, which transforms a time-domain signal to its frequency-domain form.

We first profile the FFT application to find out which routine is the most frequently executed. Then, we place that routine into the on-chip scratch memory to guarantee the fetch hit and avoid the conflict with other instructions. After the profiling, we found that the floating point multiplication subroutine is the most frequently executed, and its size is 906 bytes. For simplicity, we place the whole subroutine in the scratch memory.

The following memory hierarchy configurations are used to perform the experiments:

$1K\ scratch\ memory + 1K\ instruction\ cache$ versus $2K\ instruction\ cache$.

$1K\ scratch\ memory + 2K\ instruction\ cache$ versus $4K\ instruction\ cache$

We name the memory hierarchy with scratch memory as $new\ \ architecture$, while the memory architecture consisting of pure cache is named $traditional\ \ architecture$. The size of the scratch memory and cache determine the total amount of instructions that can be stored on-chip. Since the length of instruction is two bytes, the depth of the scratch memory and the cache is given by their respective sizes divided by two.

*3) Experimental results and discussion:* The experimental results are shown in Table I. For the first pair of configurations, the new architecture provides a faster execution time (90.67sec vs. 91.26sec), while the cache miss rate is slightly higher (17.29% vs. 15.98%). As for the second pair of configurations, the execution time (84.47sec vs. 82.02sec) and the miss rate (10.17% vs. 7.60%) of the new architecture are slightly worse than that of the traditional architecture.

TABLE I
EXPERIMENTAL RESULTS

| | $Cache\ miss\ rate(\%)$ | $Execution\ time(sec)$ | $Storage\ (kbits)$ |
|---|---|---|---|
| 2K cache | 15.98 | 91.26 | 31 |
| 1K cache + 1K scratch memory | 17.29 | 90.67 | 24 |
| 4K cache | 7.60 | 82.02 | 60 |
| 2K cache + 1K scratch memory | 10.17 | 84.47 | 39 |

The preliminary experimental results show that the new architecture can provide comparable performance as the traditional architecture. In addition, the new architecture has promising potential.

First, the new architecture consumes less memory resources. The new architecture of 1K cache & 1K scratch memory consumes an amount of 24 Kbits storage, while the resource consumption of the traditional architecture of 2K cache is equal to 31 Kbits. For the second pair of configurations, the resource consumed by the new and the traditional architecture are 39 Kbits and 60 Kbits, respectively.

Second, in this case study, the granularity of the program partitioning is at the procedure level, which may result in rarely executed basic blocks being assigned to the scratch memory. If we manipulate the program at the basic block level, more frequently used portions of the program can be placed in the scratch memory, and thus the performance can be improved.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we first present the background and motivation of our research work. Then we presented a survey of existing work in this research area. We present a case study on a new software-controlled instruction memory hierarchy, and obtained preliminary experimental results. We believe that the new instruction memory hierarchy is promising compared to the traditional instruction memory architecture.

In the future, we would like to explore several important and interesting issues.

***1. Development of profiling and instruction partitioning techniques***

First, we will continue to work on our proposed instruction memory architecture. We plan to perform the profiling and instruction partitioning at the basic block level rather than the procedure level. We are also going to study and devise suitable algorithm to perform the instruction partitioning.

***2. Co-optimization and co-synthesis of memory hierarchy and application***

The second issue is the co-optimization and generation of both the memory hierarchy architecture and the applications. We would like to fully implement our co-design framework. The framework takes the hardware constraints and applications as inputs, and outputs both the memory hierarchy architecture descriptions and the coupled optimized applications, where the memory hierarchy descriptions (e.g. Verilog specifications) can be synthesized into hardware.

During the exploration, the framework should try to make full use of all the hardware resources to avoid the performance bottleneck. For example, if the computational capability of the processor is very powerful, the memory hierarchy may become a major bottleneck. Thus, more hardware resources should be used for designing the memory hierarchy. In other words, the memory hierarchy design should take into account of the parameters of the other components of the hardware platform.

***3. Dynamically reconfigurable memory hierarchy***

The third issue we would like to explore is the reconfigurable memory architecture for embedded systems. There are

opportunities to design other kinds of reconfigurable memory architecture, such as the ability to dynamically reconfigure the line size of the cache. The applications may be able to change the configuration of the memory hierarchy dynamically during different execution phases.

***4. Mutual effects of application optimization and memory architecture***

The last issue is to study how the application optimization and the memory architecture affect each other. The optimization and the architecture are tightly coupled with each other. For example, by performing loop merging, the intermediate data buffer size may be reduced, which means that a smaller data cache size is needed. On the other hand, the loop merging increases the kernel size, and thus a larger instruction cache size may be needed. Given a fixed amount of storage resource, the problem is how to divide it into appropriate sizes of instruction cache and data cache, and perform loop transformations simultaneously to maximize the performance.

## REFERENCES


[1] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. *Memory Issue in Embedded Systems-on-chip: Optimization and Exploration*. Kluwer Academic Publisher, 1999.

[2] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their applications to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–224, 2000.

[3] Ying Zhao and Sharad Malik. Exact memory size estimation for array computations without loop unrolling. In *Proceedings of the 36th ACM/IEEE conference on Design Automation*, pages 811–816, 1999.

[4] Eddy de Greef, Francky Catthoor, and Hugo De Man. Array placement for storage size reduction in embedded multimedia systems. In *International Conference on Application-Specific Systems, Architectures, and Processors*, 1997.

[5] Niklas Petterson. A summary of storage allocation for embedded processors. 2004.

[6] Sven Wuytack, Jean-Philippe Diguet, Francky V. M. Catthoor, and Hugo J. De Man. Formalized methodology for data reuse: Exploration for low-power hierarchical memory mappings. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4), December 1998.

[7] Iroshi Nakamura, Masaaki Kondo, Taku Ohneda, Motonobu Fujita, Shigeru Chiba, Mitsuhisa Sato, and Taisuke Boku. Architecture and compiler co-optimization for high performance computing. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2002.

[8] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publisher, 1998.

[9] Preeti Ranjan Panda, Nikil D.Dutt, and Alexandru Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1), Jan 1999.

[10] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 136–146, 2003.

[11] S. Wuytack, F. Catthoor, G. De Jong, and H.J. De Man. Minimizing the required memory bandwidth in vlsi system realizations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(4), 1999.

[12] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alterative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002.

[13] Masaaki Kondo, Hideki Okawara, and Hiroshi Nakamura. Scima: Software controlled integrated memory architecture for high performance computing. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 2000.

[14] Y.N Srikant and P.Shankar. *The Compiler Design Handbook*. CRC Press, 2003.

[15] M.van Swaaij, F.Franssen, and D.De Man. Modelling data and control flow for high-level memory management. In *Proceedings of 3rd ACM/IEEE Europe Design Automation Conference*, pages 8–13, 1992.

[16] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1998.

[17] P.R. Panda, F.Catthoor, N.D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P.G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2), April 2001.

[18] Karl Pettis and Robert C.Hansen. Profile guided code positioning. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[19] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[20] Nios development board reference manual, stratix edition. In *www.altera.com/literature/manual/mnl_nios_board_stratix_1s10.pdf*.