Effective Algorithms for Partitioned Memory Hierarchies in Embedded Systems

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Jason D. Hiser

May 2005

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

Jason D. Hiser

Approved:

Jack W. Davidson (Advisor)

Kevin Skadron (Chair)

David E. Evans

Ronald D. Williams

John C. Knight

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

April 2005

Abstract

Many architectures today, especially embedded systems, have multiple memory partitions, each with potentially different performance and energy characteristics. To meet the strict time-to-market requirements of systems containing these chips, compilers require retargetable algorithms for effectively assigning values to the memory partitions. Furthermore, embedded system designers need a methodology for quickly evaluating the performance of a candidate memory hierarchy on an application without relying on time-consuming simulation.

This dissertation presents algorithms and techniques to effectively meet these needs. First, EMBARC is presented. EMBARC is the first algorithm to realize a comprehensive, retargetable algorithm for effective partition assignment of variables in an arbitrary memory hierarchy. It supports a wide variety of memory models including on-chip SRAMs, multiple layers of caches, and even uncached DRAM partitions. Even though it is designed to handle a wide range of memory hierarchies, EMBARC is capable of generating partition assignments of similar quality to algorithms designed for specific memory hierarchies. A large range of benchmarks and memory models is used to demonstrate the effectiveness of the EMBARC algorithm. Experiments show optimal or near-optimal results for every catagory of memory hierarchy tested.

This dissertation also presents MPRES. MPRES is an algorithm to estimate the effectiveness of the memory hierarchy for a given application without requiring time-consuming simulations. To show that MPRES generates accurate performance estimates, MPRES performance estimates are compared to detailed simulation results. Experiments show

performance estimations trend the same as values obtained via simulation. Furthermore, MPRES is significantly faster than simulation, requiring two orders of magnitude less time.

Together, MPRES, EMBARC, and their supporting framework provide a comprehensive solution for embedded software designers who must choose a suitable partitioned memory hierarchy and application programers who rely on the compiler to automatically assign variables to memory partitions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in information technologies have enabled computing devices to be *embedded* in a wide variety of physical devices. Wolf defines an embedded system as "any device that includes a programmable computer but is not itself intended to be a general-purpose computer" [54]. Embedded computing devices, indeed, have become integral to making modern life easier and more productive. Because of the high demand and competitiveness of the marketplace for many of these embedded devices (especially consumer devices such as cellular telephones, digital cameras, DVD players, etc.) manufacturing these devices as cost-effectively as possible is critical for marketplace success. In fact, some product volumes are so high that it is cost effective to design special chips specifically tailored for the product.

Consequently, the design of embedded systems is an important activity. Designers spend significant time balancing the cost, energy consumption, and performance of an embedded system. One significant challenge in designing an embedded system is the development of a memory hierarchy that provides high bandwidth and low latency while meeting power and cost objectives. Current trends in CPU and memory speeds complicate this design activity. For example, CPU speeds are reaching 3GHz while memory speeds are in the 100 MHz to 300 MHz range [1, 43]. Indeed, if current performance trends continue, future systems may take hundreds of thousands of CPU cycles to satisfy a cache miss. The growing disparity between processor and memory speeds has led some researchers to assert that system performance will soon hit the "memory wall" and there are no good solutions in

sight [57]. In other words, system performance will be dictated by the performance of the memory hierarchy.

Embedded processors, including application specific integrated chips (ASICs), may especially need to deal with memory latency issues. Such processors often have high bandwidth needs yet strict cost limitations. Cellular phones need to encode and decode the digital signals sent to and from cellular towers in real time. Digital camcorders need to compress and decompress video streams in real time for recording and playback. In these cases, streams of input need to be manipulated within strict time limits. If a processor is incapable of meeting the application demands, it is unsuitable for the device. In many cases, including the cell phone and digital camcorder cases mentioned previously, battery lifetime must also be extended. Consequently, memory hierarchy energy consumption is also an important consideration.

Fortunately, unlike general-purpose computer systems, an embedded system only needs to perform well on a limited set of applications. An embedded system's smaller set of target applications allow unique approaches to solving the memory wall problem. One approach is to divide the processor's memory into many separate physical units. Memory partitioning has many significant benefits. This strategy yields a memory system that allows accesses to be handled concurrently. By satisfying multiple memory accesses in parallel, a partitioned memory provides higher effective bandwidth to the processor. Energy can be reduced by accessing smaller memories (smaller memories consume less energy per access) or using a lower power mode when a partition is unused. Placing variables that conflict in a cache in separate partitions provides higher cache hit rates. However, and perhaps most importantly, partitioning memory can effectively reduce the average memory access time by satisfying multiple memory accesses in parallel. With such significant benefits to memory partitioning, it is not surprising that designers devote considerable time to determining the most appropriate partitioned memory hierarchy for their chip and target applications.

Time-to-market is an important consideration for embedded processors, embedded systems and end products. If the processor and software development tools are not released

quickly enough, a competitor's processor may be used in the end product instead. Consequently, embedded processor engineers need efficient methods for designing a memory hierarchy which meets design criteria. Designers must quickly evaluate which memory hierarchy, and what amount of memory partitioning is appropriate for their target applications. To do this, a compiler tool chain is needed to compile the application for candidate memory hierarchies. Since the compiler must assign each variable in the program to a memory partition, an algorithm to make these partition assignments is also necessary. Having a retargetable partitioning algorithm (coupled with a retargetable compiler) allows designers to quickly recompile the target applications for performance evaluation on a variety of memory hierarchies.

A retargetable partitioning algorithm allows the system designer to quickly compile the program for the target memory hierarchy, but does not help the designer evaluate how effectively the memory hierarchy satisfies the designer's performance goals. Unfortunately, the search space is large and hundreds or thousands of memory hierarchies may need to be evaluated. Consequently, the embedded systems designer needs a methodology for efficiently evaluating how effectively each memory hierarchy performs once the program's variables are assigned to partitions. It is infeasible to fabricate many unique processor dies, and simulations are many times slower than native execution. Evaluating hundreds or thousands of memory hierarchies with simulations could take weeks, or even months. This would slow the design time significantly. In order to quickly evaluate each possible memory hierarchy, an estimation technique to evaluate the performance of the partitioned application is also needed. Together, a retargetable memory partition assignment algorithm and a technique to estimate the memory hierarchy performance can help embedded processor designers find a partitioned memory hierarchy that meets design constraints.

1.1 Problem Definition

Partitioned memory hierarchies can help embedded systems meet their energy objectives, performance needs, and design-time limitations while reducing the cost of the system. For partitioned memory hierarchies to be fully exploited, compiler technology must be able to effectively assign variables to memory partitions. Furthermore, the compiler must be able to relate the effectiveness of the assignment to the system designer.

To address the issues described in the previous section, a highly retargetable memory partitioning algorithm that can generate code to effectively exploit a partitioned memory hierarchy was developed. More formally, given a memory hierarchy, H, the access characteristics of each cache, $c \in H$, each memory partition, $m \in H$, and a program, P, in a high level language such as C, the algorithm determines an assignment of variables, $\forall v \in P$ each is assigned to exactly one $m \in H$. Figure 1.1 illustrates the problem. The arrows from the program variables to the partitions represent the output of the algorithm, an assignment of variables to partitions. Since many assignments are feasible, the most desirable solution

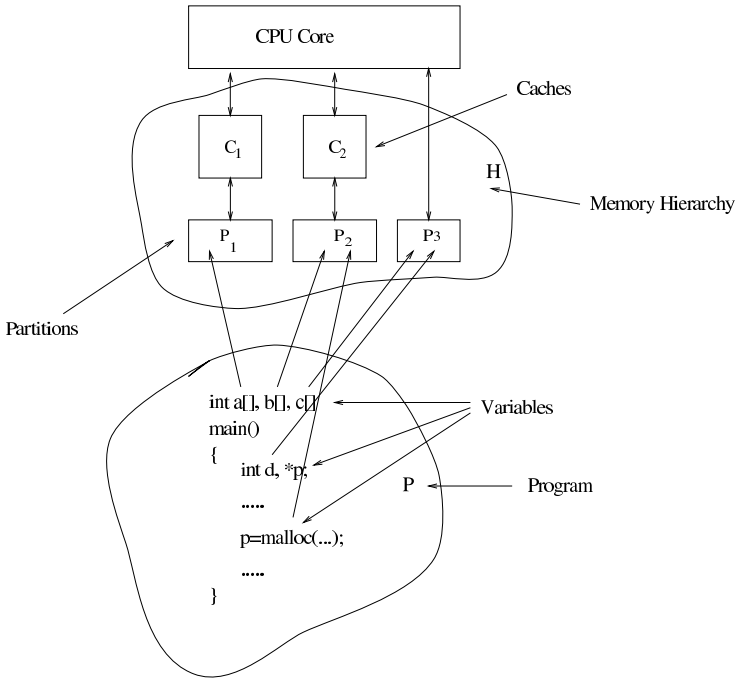

Figure 1.1: Problem Visualization

is the one in which an objective function (such as energy consumption or run time of P) is minimized.

Furthermore, to expedite the design of embedded systems with partitioned memory hierarchies, a process for efficiently evaluating the effectiveness of the memory partition assignments was developed. More formally, given a program and its input, P, and a memory hierarchy, H, an algorithm that outputs an estimated run time for P was developed.

1.2 Problem Solution

This dissertation provides an effective algorithm to assign program variables (global, stack, and heap) to the partitions of a memory hierarchy. This algorithm is named EMBARC (**E**ffective **M**emory **B**ank **A**ssignment algorithm for **R**etargetable **C**ompilers). An off-line profile of the program's data accesses is generated using a simulator, or an instrumented binary. A retargetable compiler uses the generated profile and the program source to assign variables to partitions. A greedy algorithm assigns variables from most frequently accessed to least frequently accessed, based on the off-line profile. To assign a variable, the cost of placing the variable in each partition is calculated. The cost is based on the variable's estimated conflicts in the partition, the partition's capacity, and the partition's average access times. The variable is assigned to the partition where the variable has the lowest cost. The algorithm's effectiveness is compared with algorithms designed for specific memory hierarchies. Results indicate that the retargetable algorithm provides solutions of similar quality as the dedicated algorithms, often providing optimal solutions.

This dissertation further develops and evaluations an algorithm to accurately estimate the performance of a memory hierarchy. This algorithm is named MPRES (**M**emory **P**e**R**formance **E**stimation **S**ystem). The algorithm uses the data profile to estimate the miss rate of each cache in the memory hierarchy. Using the estimated miss rates, the average access time of each cache is calculated in a top-down manner. Next, each partition's average access time is estimated. Finally, the average time for the program to access mem-

ory is calculated. The results of the algorithm are compared to simulated results, and the results demonstrate that the trends match very closely.

1.3 Contributions

This dissertation develops algorithms to help embedded system designers and compiler writers effectively cope with the challenges faced when using partitioned memory hierarchies. In particular, this dissertation makes the following major contributions:

- A technique is presented for collecting data profiles from a program in which each variable has a list of pseudo-live ranges.
- A memory hierarchy description language is presented. The language is suitable for describing the memory hierarchy of a machine in enough detail that partition assignment and runtime estimation are possible, but simple enough for embedded system designers and compiler writers to use without undue burden.
- Furthermore, a technique, called EMBARC, for assigning program variables to memory partitions given a memory hierarchy description and the program is presented. Previous techniques have assumed a fixed memory hierarchy. This assumption severely limited the use of previous algorithms.
- Next, MPRES, a technique to estimate how effectively a memory hierarchy satisfies requests for a given application, is presented. MPRES can be used to quickly evaluate a large set of memory hierarchies for a set of target application. Previously, embedded system designers relied on costly simulation to gather such information.
- Lastly, EMBARC and MPRES are evaluated using a comprehensive benchmark suite and a wide variety of memory hierarchies. This evaluation demonstrates the value and effectiveness of EMBARC and MPRES in selecting suitable memory hierarchy for embedded systems. This evaluation also demonstrates the need for suitable benchmarks when evaluating memory hierarchy algorithms.

1.4 Organization

The remainder of this work is organized as follows: Chapter 2 discusses related work. Chapters 3 and 4 describe the memory partitioning algorithm and the partition assignment evaluation algorithm, respectively. Chapter 5 evaluates its effectiveness. Finally, Chapter 6 summarizes the contributions of this work and presents directions for future research.

Chapter 2

Background and Related Work

This chapter provides background information and discusses previous research in memory partitioning and performance estimation. Section 2.1 discusses alternate approaches to addressing the memory wall problem and their shortcomings, while Section 2.2 discusses the details and tradeoffs of memory partitioning. Previous memory partitioning research and its limitations is discussed in Section 2.3. Section 2.5 gives background on previous performance estimation techniques and their shortcomings for partitioned memory hierarchies.

2.1 Alternate Approaches to the Memory Wall Problem

Architects have used a number of techniques to satisfy the memory access requirements of high performance processors. This section discusses those techniques, their strengths, and their shortcomings.

2.1.1 Increasing Register File Size

One easy and obvious approach for avoiding off-chip accesses is to make more registers visible to the software. The benefits of using more registers was brought to light with the invention of RISC computers which frequently have more registers than CISC computers [12]. Unfortunately, adding additional programmer-visible registers can have detrimental side effects. One of the major disadvantages is that large register files consume much die space

due to necessary multi-porting [39, 17, 16, 18, 29]. This may lengthen cycle time. Even if larger die space and higher cycle times are not critical issues, the size of an instruction must grow in order to accommodate the register encoding. In turn, larger instructions increase the bandwidth necessary in the fetch engine of a chip, complicate decoding and register renaming, and increase issue logic. Another problem with large register sets is that they are often not dynamically addressable. Without dynamic addressability, register's usefulness is limited to scalar variables that are known to be alias-free. Lastly, the operating system must execute code to explicitly save and restore these registers on a context switch—a potentially costly process [38].

2.1.2 Increasing Cache Size and Associativity

Increasing cache size is another way to provide better effective memory system performance, but this approach is reaching the limits of effectiveness. As caches become larger, they also become slower. The Compaq 21264 chip and Sun's UltraSPARC-III already employ pipelined data and instruction caches [32,27]. To increase cache sizes, deeper pipelining and longer delays will be necessary. Another problem with increasing cache size is that a point of diminishing returns is quickly reached. Even if the cycle time can be maintained without deeper pipelining, doubling the size of the data cache may result in only minimal speedup because programs do not exhibit enough locality. For example, Hennessy and Patterson report that moving from a 32KB direct mapped data cache to a 128KB data cache results in only a 2.02% reduction in cache miss rate for the SPEC92 benchmark suite [26].

Manufacturers have also moved away from direct-mapped caches to set-associative caches [51, 50]. This reduces cache misses that occur due to conflicts. Yet, studies have shown that while small amounts of associativity (2-4 way) can help significantly, larger amounts (8-way or more) often have little or no benefits. As an example, a 32KB data cache shows a reduction in miss rate of 0.6% when it is changed from direct mapped to 2-way associative for the SPEC92 benchmark suite. However, only a 0.1% reduction is seen when moving from 2-way to 8-way associative [26]. Even if full associativity were feasible

within cycle time constraints, cache misses still occur when the capacity of the cache is reached or when a new working set becomes active.

These issues are discussed more fully by Hennessy et al. [26].

2.1.3 Cache Management Schemes

Compiler writers have worked hard to understand exactly how caching affects load/store latency. It has been shown that many cache misses are predictable and using some instructions to give the cache "hints" about future execution is useful. These hints can be of many forms, but one of the most popular is the prefetch instruction [33, 21, 34, 4]. Cache prefetch instructions instruct the cache to fetch a value from memory because it is likely to be used soon, potentially avoiding cache misses.

This technique performs well, but has some drawbacks [26]. First, the compiler must recognize which memory instructions are likely to miss in the cache. If the compiler is too aggressive when inserting prefetch instructions, wasted fetch, decode, and issue bandwidth can hurt performance. On the other hand, if the compiler directs the cache to prefetch a data item and that data item is not subsequently used, the resulting cache pollution can hinder performance.

A slightly different approach is to "fetch around" the cache [19, 37]. In this approach, the compiler or hardware detects that a value has little locality and is not a good candidate for the cache. With fetch-around, a value fetched from memory is delivered directly to the appropriate execution unit and the value is not placed in the cache. Fetch-around avoids polluting the cache with values that have low locality. Although this approach can help keep high-locality variables in the cache, and consequently make better use of those resources, it does not avoid the ever-growing CPU/memory latency issue.

A third approach is to use a *victim cache* [30]. In this approach, when a cache line is evicted from the cache, the line is not directly written to memory, but is instead sent to the victim cache. Subsequent cache misses check the fully-associative victim cache before a memory access. A victim cache is effective for cases where the cache does not have enough

associativity to hold the working set. The victim cache effectively gives the cache more associativity at lower cost and cycle time than a cache with higher associativity, but it does not address the latency of cache miss time or reducing off-chip context.

2.2 Memory Partitioning

Section 2.1 discussed approaches to reducing the average memory latency to the CPU that have provided results in the past. In order to continue to reduce average memory latency, memory partitioning is necessary. This section discusses background information and related research in the area of memory partition assignment.

In a *partitioned memory architecture,* two or more memory partitions are compiler-visible. Figure 2.1 contains a sample partitioned memory architecture. In the figure, the execution core can access either DRAM via the primary data cache, or access an on-chip SRAM.

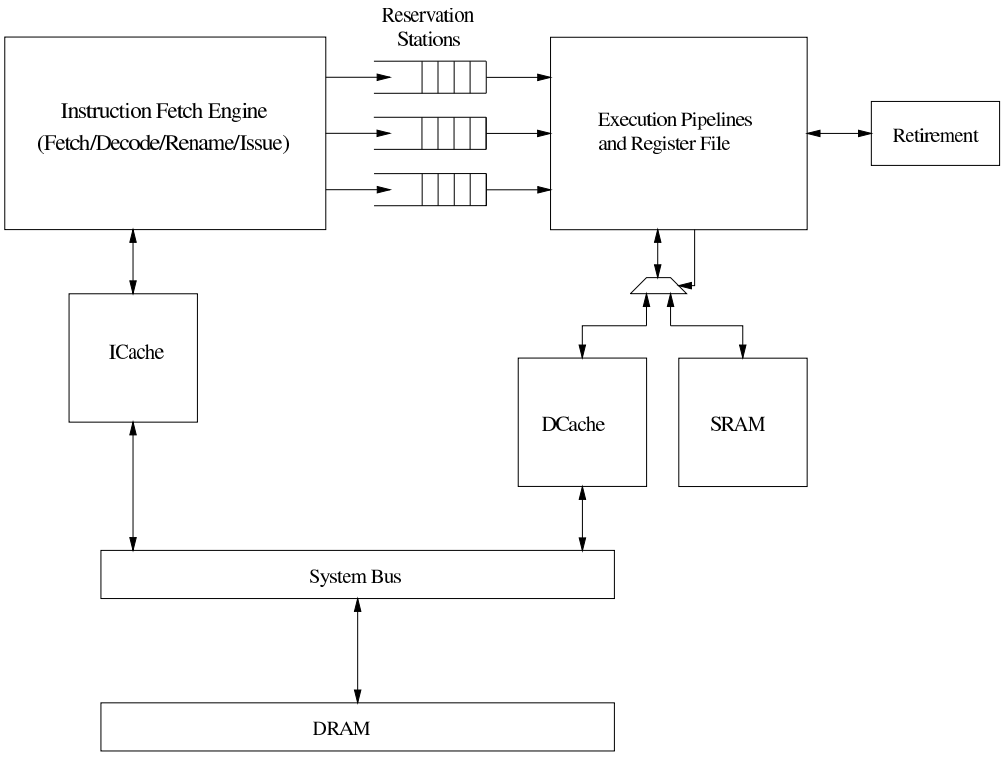

Figure 2.1: Sample partitioned memory architecture

2.2.1 Benefits

There are many significant advantages to partitioning the memory hierarchy. The hardware can satisfying multiple memory in parallel, consequently running the application faster. Consider again Figure 2.1. By separating the on-chip resources into data cache and SRAM, the execution core can satisfy a memory request to each partition in parallel. Consequently, the program would execute faster.

Furthermore, energy can be reduced by accessing smaller memories. Again, Figure 2.1 shows that a data access can be satisfied by the data cache or an SRAM. Since the SRAM has no tags, valid bits, or dirty bits, access to the SRAM consumes less energy than accesses to the equalize-size data cache. Furthermore, the SRAM reduces the number of accesses to the off-chip DRAMS or off-chip caches, further reducing energy consumption.

Finally, the configuration offers more flexibility because the processor can enable faster memories at the cost of higher energy consumption, or disable them to run in lower power modes. Again consider Figure 2.1. If a program is designed to run on the processor for the lowest possible power, perhaps at the cost of slower execution, the program could specify that the SRAM is to be disabled. It would then be illegal for the program to access the SRAM, and the contents of the SRAM would be lost, but the energy consumption of the processor would be minimized. Such an operating mode would be ideal on a mobile processor which is mostly idle, but still needs to perform background calculations.

2.2.2 Drawbacks

Although memory partitioning is generally beneficial, there may be drawbacks to partitioning memory. First, there is a hardware cost to partitioning. As Figure 2.1 shows, extra logic is needed to determine which partition to access. The more partitions in the architecture, the more expensive this logic becomes. The benefits of adding another partition must outweigh the cost of the additional logic needed to access the new partition.

Second, if the application is not effectively exploiting the available hierarchy, the costs associated with the partitioning are not sufficiently amortized. Again consider Figure 2.1,

if a program does not make sufficient use of the SRAM, it might have been better to devote all the on-chip resources to the data cache. Devoting the on-chip resources to cache would lower the miss rates, and make the program run faster.

To ensure that the benefits of partitioning outweigh the costs, it is necessary to make sure each program effectively uses the given memory hierarchy. Compiler techniques are needed to ensure the memory hierarchy is used effectively.

2.2.3 Types of Memory Partitioning

There are two main approaches to exposing the memory partitioning to the compiler. One approach, the *ISA paradigm*, the memory partition to be accessed is specified by bits in memory accessing instructions. In the *address paradigm*, the variable's address specifies which partition to access. Sections 2.2.3.1 and 2.2.3.2 discuss these models further.

2.2.3.1 The ISA Paradigm

Perhaps the simplest approach for incorporating a partitioned memory hierarchy into a system is to augment the instruction set with special load and store instructions. We call this technique the *ISA paradigm*. In this approach, there are load and store instructions corresponding to each partition. These instructions are fetched, decoded and issued in the standard way, except that instead of accessing a unified load/store queue (LSQ) of the cache hierarchy [6, 42, 49], they access an LSQ corresponding to instruction's partition. Partitioning the LSQ helps distribute the memory traffic. Also, since loads and stores from one partition are guaranteed not to be aliased with memory accesses from another partition, the hardware to perform memory disambiguation is simplified.

To use the hardware effectively, the compiler needs to identify variables that are alias free. All variables that may have aliasing need to be assigned to the same partition in order for the compiler to generate correct code. Consider the code in Figure 2.2. If variables `a` and `b` are assigned to different partitions, then the compiler cannot possibly choose a load

instruction for the reference in function `sum` that is correct. Thus, only alias-free variables are free to be assigned to a partition independently of other variables.

```
char a[100];
char b[100];

int main()
{
   ...
   ... = sum(a);
   ...
   ... = sum(b);
   ...
}

int sum(int p[])
{
   for(i=0; i<100; i++)
        sum += p[i];
   return sum;
}
```

Figure 2.2: Code demonstrating partition assignment constraints in ISA paradigm.

Given this alias-free constraint, the compiler may not be able to assign some variables to the best partitions due to aliasing. For example, a variable with a short lifetime and poor locality that would otherwise be a good candidate for fast on-chip memory may have to reside in slower DRAM because of aliases. Such a situation could result in poor utilization of the memory hierarchy.

2.2.3.2 The Address Paradigm

Even though the hardware for the ISA paradigm is simple to implement, it may not yield satisfactory results because of the variable assignment restrictions due to aliasing. A more aggressive approach could set aside portions of the (virtual) address space for each partition. This paradigm, called the *address paradigm*, is easily implemented in hardware as well, but complicates LSQ implementation. The hardware implementation of this approach uses no

special instructions. Instead, any memory instruction can potentially access any partition. All memory instructions share an LSQ, and the appropriate memory is accessed only after the effective address is calculated.

The address paradigm potentially makes memory address disambiguation more difficult than the instruction paradigm, yet it is no more difficult than what is seen in a standard processor. This is true because on a conventional processor, all memory accesses are satisfied by a single LSQ. The only additional hardware required is that which determines which partition is accessed once the effective address computation is complete.

The benefit of this approach is that the compiler can safely assign any variable to any partition, even in the presence of aliases. The compiler can consequently assign arrays and structures to any partition across procedures (that are compiled separately), as well as global variables.

Because of the potential drawbacks of the ISA paradigm and the extra compiler complexity that is needed, the remainder of this work focuses on the address paradigm.

2.3 Memory Partitioning Research

There are many algorithms which perform memory bank assignment for specific categories of embedded systems. Previous research can be broadly categorized into those that choose a static partition assignment for each variable, and those that change the partition assignment during runtime of the program. Sections 2.3.1 and 2.3.2 discuss the previous research in further detail.

2.3.1 Static Partition Assignments

With static partition assignment the compiler chooses a single partition for each variable at compile time. This assignment remains in effect for the lifetime of the program. Since this technique is easy to understand and incurs no runtime overhead, static partitioning assignment algorithms for specific memory hierarchies has been studied extensively [44, 22,

41, 40, 10, 7, 8]. Research has focused on multibank cacheless systems for higher bandwidth (Section 2.3.1.1), cacheless systems with SRAM (Section 2.3.1.2) and cached systems with SRAM (Section 2.3.1.3.)

2.3.1.1 Multibank Cacheless Systems

Multibank cacheless systems are common when system designers or application programmers decide that predictable execution characteristics are needed, but still require high data bandwidth. To meet these bandwidth requirements, multiple banks of memory are added and the CPU is allowed to access separate banks in parallel. Figure 2.3, taken from Saghir, et al., illustrates how such a system may function [44]. In the figure, a standard pipeline (a VLIW pipeline in this case) references two different DRAM banks, without any caches.

Both Saghir and Delaluz recognized the need to support these systems and provided solutions to reduce execution time and energy consumption [44, 22]. Saghir's work focuses on generating code for machines like the Motorola DSP56001. The Motorola DSP56001 is a VLIW machine which can execute two memory operations in a single instruction, provided that they access separate memory banks. In order to efficiently generate code, Saghir's method couples packing the VLIW operations with the partitioning of memory. The partitioning algorithm closely mimicked the operation of a list scheduler. When the scheduling algorithm found two memory operations that could occur in parallel, the cost of keeping the accessed variables in the same partition was increased. An interference graph holds the costs associated with partitioning. After the scheduling phase, a graph partitioning algorithm is used to partition the interference graph, effectively assigning variables to memory partitions. The code is then scheduled with the partition assignments in place.

Delaluz describes a technique to partition memory in order to maximize the time that memory partitions are idle [22]. Putting the idle partitions in a lower power mode makes the technique capable of significant power savings. Delaluz's technique first performs a loop fission and loop splitting to group array accesses that have similar patterns together [9]. Then, the allocation phase attempts to maximize the idle time of memory partitions by

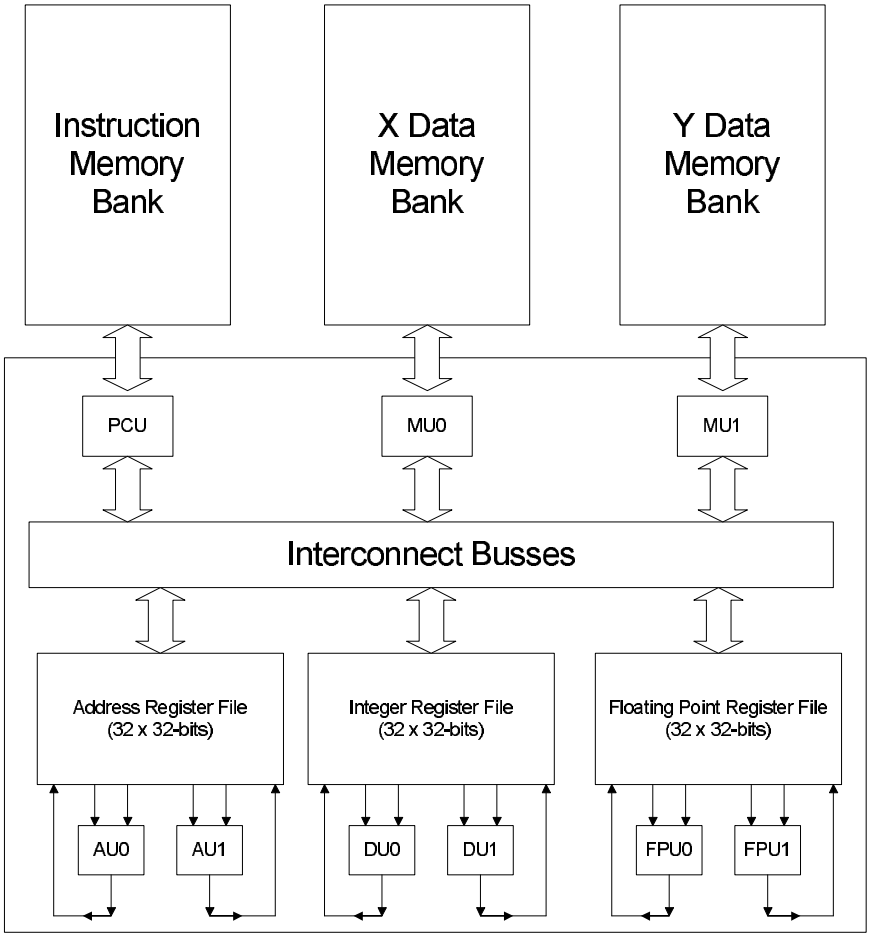

Figure 2.3: Sample system with 2 DRAM banks

placing variables that are accessed together into the same partition.

Unfortunately, both techniques assume that there are no data caches in the system. Since each technique relies heavily upon this assumption, there is no easy way to generalize these techniques to include caches.

2.3.1.2 Cacheless Systems with SRAM

Although multibanked cacheless systems are one common way for meeting the high bandwidth requirements of some systems, other systems need faster access to some data. For example, a graphics processing program may need to repeatedly compare a smaller image (stored in an array) to sections of a larger image. Because the pattern needs to be read many times, storing the pattern in a fast on-chip memory can save significant time. To address such problems, system designers may add very fast on-chip SRAM.

In these systems, the assembly language programmer or the compiler specifies that a variable should reside in SRAM by assigning the variable to the virtual addresses (or physical addresses in a system without virtual memory) corresponding to the SRAM. Fast, on-chip SRAM memory provides many of the advantages of caches without adding unpredictability to the system.

Avissar proposed an optimal solution for assigning variables to partitions in cacheless systems using an integer linear programming (ILP) solution [7, 8]. A linear constraint is used to represent the access time of the variable in terms of reads and writes of the variable. Further constraints are placed to ensure that the partition size is not exceeded. The linear solver is directed to find a solution that minimizes the memory access time. Since there are no caches allowed in these systems, the solution is provably optimal (ignoring data alignment and placement).

Unfortunately, the ILP solver may be too slow for larger, more complex programs. The $O(2^N)$ runtime nature of the ILP takes exponentially longer when the number of partitions or number of variables increases. In a complex program, the solver may take hours, or even days to complete.

Furthermore, programs with pointers and dynamic allocation may make the solution difficult to apply. Such program features prevent the compiler from knowing which variable is accessed by a load instruction. Even worse, with dynamic allocation, the dynamic variable's size is unknown until runtime. Consider the source code in Figure 2.4. The memory pointed to by `buffer` can be any size from 1 byte to 4k bytes. Without this knowledge, it is impossible to accurately formulate the linear equations needed by the ILP solver. Thus the optimal solution may not be computable.

```
main()
{
  char *buffer;
  int size=get_size_of_file("test.input");

  if(size<4096)
    buffer=malloc(size);
  else
    buffer=malloc(4096);

  ...
}
```

Figure 2.4: Dynamic allocation source code example

Even when the time for an optimal solution is justified and alias analysis is not an issue, there is no easy way to extend the ILP solution to machine models with caches. Thus, like the solutions by Saghir and Delaluz, these solutions are not general enough for the needs of an embedded system designer.

2.3.1.3 Cached Systems with SRAM

The advantages of having on-chip SRAM are not limited to systems without caches. SRAM, when used effectively, offers fast, energy-efficient access to data even when the system includes a cache. Figure 2.5, taken from Panda, et al., illustrates how such a system may function [41, 40].

Efficient management of the SRAM can eliminate extra reads or writes of data that

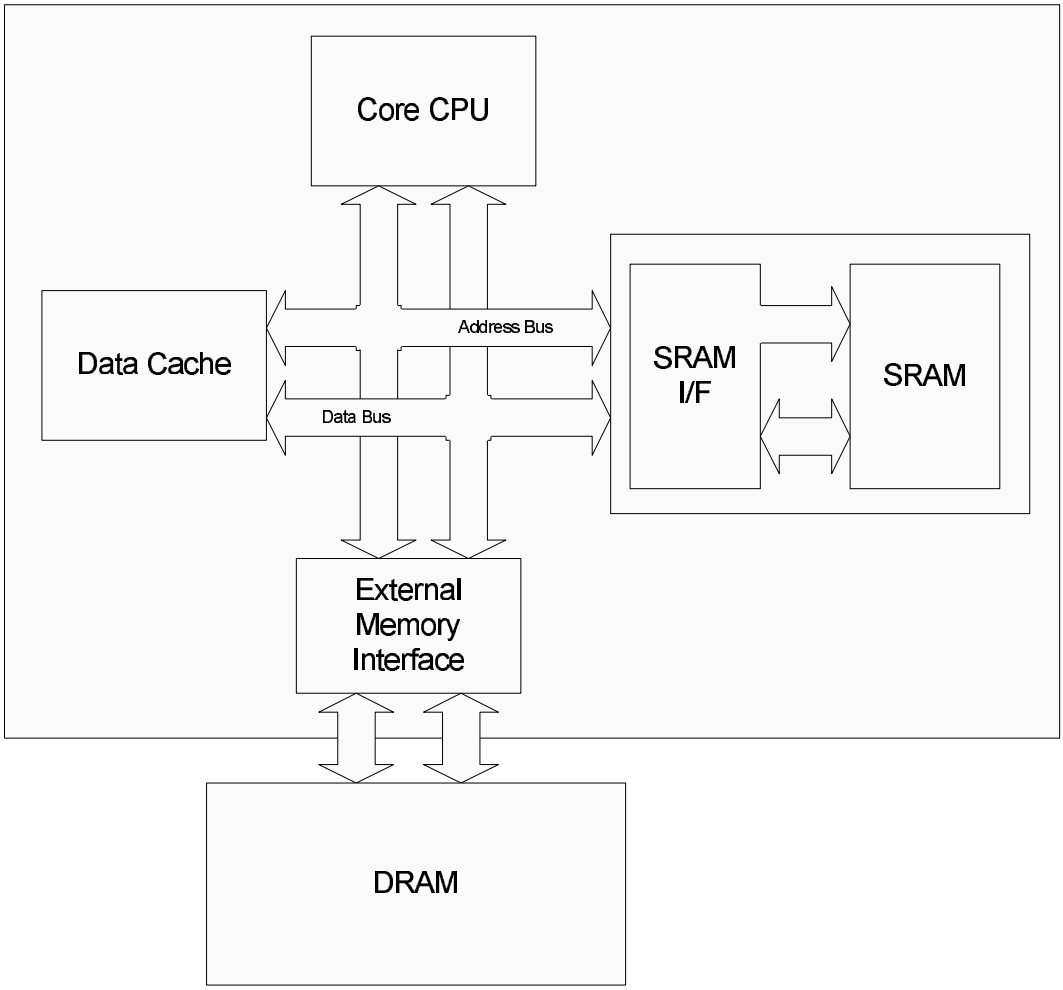

Figure 2.5: Sample system with SRAM and cache.

a cache may not be able to avoid. Furthermore, keeping the most used data close to the processor avoids coincidental evictions that a cache may suffer from. Panda and Banakar recognized the need to support automatic partitioning in these types of systems [41, 40, 10].

In Panda's research, the program variables are clustered by lifetimes [41, 40]. Variables in the same cluster have non-overlapping lifetime. Each cluster is then assigned to on-chip SRAM. The assignment algorithm is based on the 0-1 knapsack problem. It treats each cluster as an item to put into a knapsack; the knapsack size is the SRAM size. The size of the cluster, the number of accesses, and a total conflict factor are used to calculate the cost of the variable. When the knapsack (SRAM) is full, the remaining variables are placed in DRAM.

Banakar's research evaluates exchanging the processor's traditional cache for SRAM [10]. The *encc* compiler is used to generate code for processors with varying amounts of SRAM or cache. Both instructions and data are mapped to the SRAM by the compiler, which also uses a 0-1 knapsack algorithm. The target processor, an ATMEL AT91M40400 (based on the ARM ARM7TDMI), and a Cacti power model are used to estimate the energy usage of each configuration. Banakar, et al. found that systems with on-chip SRAM have as much as a 56% reduction in the die-area and cycles product, and as much as a 40% power saving over systems with a pure cache configuration [10].

However, both Panda's and Banakar's solutions are specific to memory hierarchies with a single (hierarchy) of data cache(s) and a single SRAM. If the system has more than one cache, cache bypassing, EPROM, or an off-chip DRAM partition, such solutions cannot be directly applied or easily converted.

If multiple first-level SRAMs or caches are present, these solutions are no longer feasible.

2.3.2 Dynamic Partition Assignment

Some previous work has attempted to make partition assignment dynamic; a variable's partition assignment may change as the program enters different phases. Kandemir and Udayakumaran describe solutions that move variables between SRAM and DRAM during

the execution of a program [31, 52]. Often the speed and energy benefits of being able to change assignments during execution overcomes the runtime overhead of this movement to and from the on-chip SRAM.

In Udayakumaran, et al, the compiler locates program points where transfers between SRAM and DRAM may be beneficial [52]. At each of these points, the compiler uses a heuristic to decide which variables belong in SRAM (taking into account the cost of transferring variables). The assignments remain in effect until the program reaches (dynamically) the next program point in which transfers can take place. Finally, the algorithm inserts copy instructions (in the form of DMA or actual instructions to copy the data). Udayakumaran's approach demonstrates runtime savings of 31.2% to 34.2% depending on machine model over purely static approaches.

Kandemir's approach uses a static assignment algorithm to drive a dynamic placement approach [31]. First, the partition assignments are made for the entire program. Then, a program point is chosen. The program is separated into two portions, and the static placement algorithm is run for both portions of the program. If the cost of using two separate assignments (including the overhead of transforming the assignments) is less than the single assignment technique, the new solution is used. The static components of the program continue to be separated in a divide-and-conquer method until it is found to no longer be beneficial.

Unfortunately, being able to change the address of a variable so that it can reside in the SRAM partition or in the main DRAM partition at different points in the execution of the program is a difficult problem. Advanced pointer and alias analysis are necessary to do such dynamic remapping, and it may be impossible to make dynamic assignments in a large embedded application. Previous work in dynamic partition assignment leaves these issues open.

Even if dynamically changing partition assignments is possible, making partition assignments for program phases is significantly more difficult when an arbitrary memory hierarchy is allowed. This work focuses on a method to choose static partition assignments. This is

a first step to solving the difficult problem of choosing dynamic partition assignments.

2.4 Data Layout Research

Much work has been done which chooses the best order and alignment of variables, or fields within a variable [15]. Data layout work is complementary to the work presented here. One could use EMBARC to choose which partition a variable belongs in, then use a data layout algorithm on each partition in the system to decide the alignment of the variables assigned to the partition. This would provide further benefits than using either technique alone.

2.5 Performance Estimation Research

Assigning variables to memory partitions in a partitioned memory hierarchy is indeed important to effectively taking advantage of the memory hierarchy. Assigning variables effectively, however, does not enable an embedded system designer to determine the effectiveness of the memory hierarchy for a target application. To determine the effectiveness of a memory hierarchy, a system designer may need to rely on time-consuming simulations. Since time-to-market is such a demanding constraint, system designers might rely instead on quicker methods for estimating the performance of the memory hierarchy. This section discusses previous research in performance estimation.

Jacob et al. describe cache models that address memory hierarchy design for generic workloads [28]. Jacob's technique uses a parameterized workload, and estimates the performance of a memory hierarchy. He validates his technique using actual simulations. Unfortunately, the technique cannot be applied to a single application, instead it only works for a range of workloads. Thus, such models are not applicable to ASIC-style systems because in an ASIC the workload is fixed by the application. ASICs do not need to perform well on a wide range of workloads, like a desktop machine does.

Other work focuses on statically predicting the cache behavior for each kernel loop of a program. Ghosh's research focuses on predicting the cache miss rate for kernel loops in a

program [23]. The compiler identifies memory accesses and uses calculus to determine the cache miss rate based on the array indices.

Unfortunately, some applications, such as *dijkstra*, cannot be analyzed by static techniques because of function calls, pointers, and dynamic allocation. Further, many static techniques require that array indices are affine functions, or operate only on kernel loops, not whole programs.

2.5.1 Faster Simulation Research

Other work aims to solve the memory hierarchy evaluation problem in a different way—speeding up simulation time. Reducing simulation time can be achieved by simulating multiple caches in one simulation run, or by taking advantage of cache properties such as the cache inclusion property or how associativity affects caches [56, 53].

Wang et al. introduce a technique to reduce the size of a memory trace by eliminating memory references that will not affect the state of caches [53]. Furthermore, their technique produces performance results for a variety of caches in a single run of the simulator. Hit ratios, write-back counts, and bus traffic can be obtained using their system.

Wu et al. introduce *iterative cache simulation* [56]. In their work, they propose taking advantage of inclusion properties of caches to infer whether some memory references will be hits or misses. They find that significant time can be saved in simulation of other caches by using these observations.

These techniques still require many simulations to fully examine the spectrum of possible memory hierarchies and take extensive time to complete. Also, it is difficult to reason about properties of a lower-level cache when multiple first-level caches are cached by the lower-level cache. Consider the case where having an instruction and data cache are both cached by a second-level cache. It is difficult to estimate the second-level cache hit rates when its associativity changes because the contents of the second-level cache include both data and instructions. Furthermore, changing either level-1 cache will change the interleaving of data and instruction accesses, significantly changing the accesses to the second-level cache.

Other work only addresses part of the problem, such as how to choose first-level cache parameters (size, associativity, line size, etc.) or how to choose the number of memory partitions assuming there is a first-level cache [5, 47, 36, 48]. Lastly, a single simulation run, no matter how many cache parameters are simulated, cannot take into account varying SRAM size. Varying the partition size changes which variable accesses are serviced by a cache, and can significantly change the access patterns. Thus, to get a picture of how the entire memory hierarchy performs, a chip designer may still need a very large number of simulations.

2.6 Summary of Related Work

Although previous research has addressed the need for partition assignment algorithms, there are many shortcomings in the collection of previous algorithms. First, many algorithms work only for a particular type of memory hierarchy (such as cacheless memory hierarchies with SRAM, or cached hierarchies with SRAM) and cannot easily be converted to work for arbitrary memory hierarchies. Second, some algorithms, may take exponential time to complete. As problem sizes grow (to include thousands hundreds of thousands of dynamically allocated variables), such algorithms may become infeasible. Worse yet, if a system designer is evaluating many memory hierarchies, even the collective time to solve many small problems may be infeasibly long. Third, some research assumes only global variables need partition assignment. Embedded benchmarks sometimes include dynamic allocation that must be considered when making partition assignments. Lastly, some research assumes access patterns that can be determined statically (such as array indices being affine functions.) Although static techniques may provide adequate results for small benchmarks, large benchmarks with dynamic memory allocation, intensive pointer use, and complex memory access patterns are beyond the scope of current static techniques.

As evidenced by previous research, embedded systems already rely on partitioned memory to meet their requirements. Yet there is a need for a retargetable algorithm that ef-

fectively assigns variables to memory partitions. An ideal memory partition assignment algorithm would not only provide an optimal assignment of variables to memory partitions, but meet a number of other requirements as well. First, an ideal algorithm would quickly retarget to a broad range of memory hierarchies. Retargetability allows embedded system designers to quickly experiment with a range of memory hierarchies. Recompiling the compiler or specifying the memory hierarchy in great detail would waste valuable time during the short design cycle of an embedded system. Second, an ideal algorithm would easily fit into a wide variety of compiler designs. Lastly, it needs to be an efficient algorithm. If the algorithm takes days or weeks to complete, experimenting with different memory hierarchies again becomes infeasible. To effectively obtain these goals, a partition assignment algorithm will need to assign every variable in the program to the proper memory partition. If the algorithm is not able to assign many variables, the partition assignment can be over-constrained. Furthermore, the algorithm will need information about memory accesses. This information can come from static program analysis, or dynamic program traces. Chapter 3 presents the EMBARC algorithm which addresses these problems. EMBARC is a retargetable, effective, efficient algorithm for assigning variables to memory partitions.

Previous research has also addressed the problem of evaluating the quality of a memory hierarchy. Unfortunately, past research also has many drawbacks. Some research statically predicts the performance of the kernel loops in an application. Such static predictions rely on the compiler's ability to analyze the memory reference patterns in the kernel. If the application has no clear kernel loops, or variable aliasing, or complex array indexing prohibits the compiler from thoroughly analyzing the application, any estimates would be nearly useless. Other research provides estimates for a range of workloads, and is not applicable to single applications. Some researchers have focused on reducing simulation time by taking advantage of cache properties, or by simulating multiple memory hierarchies in a single simulation. To address these problems, Chapter 4 describes the MPRES algorithm. MPRES is an efficient algorithm to estimate the performance of a memory hierarchy on a

given application.

Together, MPRES, EMBARC and their supporting framework provide a comprehensive solution for embedded system designers who must choose a suitable partitioned memory hierarchy and application programmers who rely on the compiler to automatically assign variables to memory partitions.

Chapter 3

Assigning Variables to Memory Partitions

To develop an algorithm to meet the needs presented in the previous chapter, a compiler framework in which to implement and evaluate a partition assignment algorithm is needed. The compiler framework chosen was the Zephyr infrastructure [58, 35].

To efficiently assign variables to memory partitions, EMBARC needs to be able examine the program being compiled, read a dynamic profile, and read a memory hierarchy description. EMBARC is implemented inside the Zephyr infrastructure, which provides access to the program being compiled. Profile information is gathered from a sample input run using a modified version of SimpleScalar [14]. The memory hierarchy is provided by the user in the form of a partition description language.

The chapter is organized as follows: Section 3.1 discusses how EMBARC fits into the Zephyr infrastructure. Section 3.2 describes the implementation of the EMBARC algorithm including details about the partition description language (Section 3.2.1), dynamic profiling technique (Section 3.2.2), and finally how EMBARC assigns variable to memory partitions (Section 3.2.3).

3.1 Zephyr

Zephyr is a retargetable compiler infrastructure currently supporting the C language and a variety of target machines including Sparc, the Intel i386 line, and the ARM/Thumb

embedded processor. The Zephyr infrastructure is made up of a front end, a code expander, a pre-linker named VLINK, and an optimizing backend named VPO.

The Zephyr front end is built around *lcc*[1], a simple C compiler [25]. *lcc* is responsible for preprocessing and parsing the source code, checking for proper syntax, and creating a parse tree. The code expander (CE) performs a straightforward (no optimizations) translation from the *lcc* parse tree to the low-level backend intermediate language. Both *lcc* and the CE work on individual source modules.

VLINK combines the unoptimized intermediate language from many files into a single file by pre-linking the intermediate language. Pre-linking renames labels, and symbols that are local to a file so that no name conflicts occur. Pre-linking is not strictly necessary for the Zephyr infrastructure to work properly, but it eases the implementation of whole-program optimizations (such as assigning variables to memory partitions.) The resulting pre-linked file is then passed to VPO for optimization.

VPO, or the Very Portable Optimizer, is the core of Zephyr's optimization engine. VPO is a retargetable machine-level optimizer that operates on an intermediate format called register transfer lists or RTLs [11]. RTLs are lists of operations. The operations in an RTL show which registers are set, how registers are used and whether memory items are accessed. The RTL operations define an instruction on the target machine. The RTLs do not necessarily contain information about which variables in the program are accessed. Instead, they show that memory is accessed via a particular addressing mode.

Like many compilers, VPO optimizes and emits one function at a time. VPO applies most classical optimizations at the RTL level, including register assignment, instruction selection, common subexpression elimination, strength reduction, and induction variable elimination. Because all optimizations are applied to the same intermediate format, optimizations can (and are) applied iteratively until no further improvements can be made. This makes VPO a very powerful optimizer, whose code quality rivals that of production

[1]Zephyr actually comes with two front ends, *lcc* and EDG. For the work presented in this dissertation, *lcc* was used exclusively. There is no technical reason why EDG (or any other front end) could not have been used, *lcc* was chosen because it is the most thoroughly tested front end within the Zephyr infrastructure.

compilers.

Figure 3.1 shows an example of how two files, *main.c* and *aux.c*, are compiled in the Zephyr infrastructure. Boxes represent input and output files, and circles represent Zephyr programs that operate on the files. In the figure, both source files are passed to *lcc* for pre-processing and syntax checking. The parse tree is written to a file, and the CE reads this file and produces unoptimized RTLs. VLINK combines the RTL files (*main.cex* and *aux.cex*) and produces the linked RTL file, *all.cex*. VPO optimizes the RTLs, and emits assembly code. The assembly code is assembled and linked traditionally to produce a native executable.

We chose to implement EMBARC within Zephyr. Implementing EMBARC within Zephyr is a logical choice because Zephyr is highly retargetable, allowing testing of EMBARC across a range of platforms. Furthermore, VPO is able to generate code that works with the SimpleScalar simulator [14]. Using SimpleScalar facilitates statistics collection on the final executable across a wide range of memory hierarchies. By using the VLINK utility in conjunction with VPO, EMBARC also has the ability to examine the entire program during one execution of VPO. The ability to examine the entire program allows EMBARC to make global decisions about variable placement without undue complication.

Figure 3.2 shows how EMBARC fits into the Zephyr infrastructure. Zephyr first generates an executable for a simple memory hierarchy (one partition with no caches.) The executable is used to generate a dynamic profile. The dynamic profile, the partition description, and the unoptimized RTLs are inputs to EMBARC. EMBARC generates the partition assignments, and VPO applies them to the unoptimized RTLs before optimization.

Since VPO optimizes and emits one function at a time, EMBARC must determine a memory partition assignment for each variable before the optimization process begins. To make these assignments, EMBARC collects any information it may need in the first pass (to compile the program for profiling) and stores this information for the second pass (final assembly code output). Although the optimized functions from the first compilation could be stored and partition assignments changed for the final assembly code, we instead choose

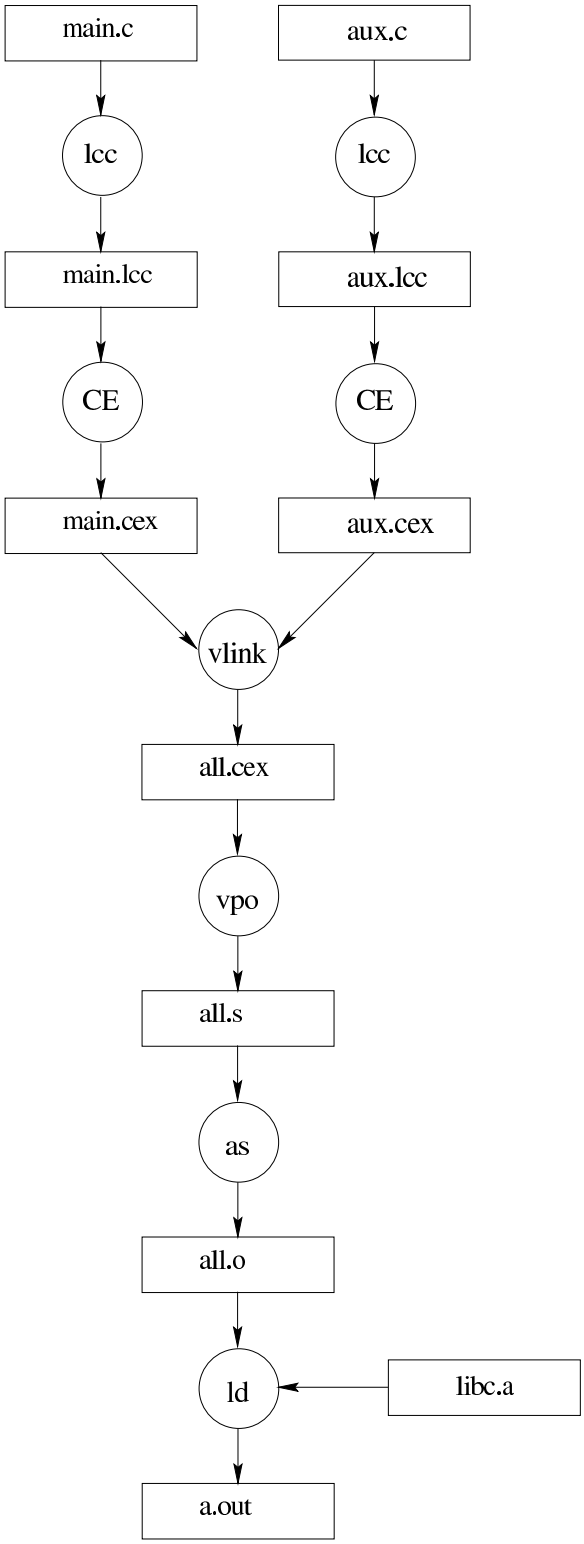

Figure 3.1: Example of how files are compiled in Zephyr

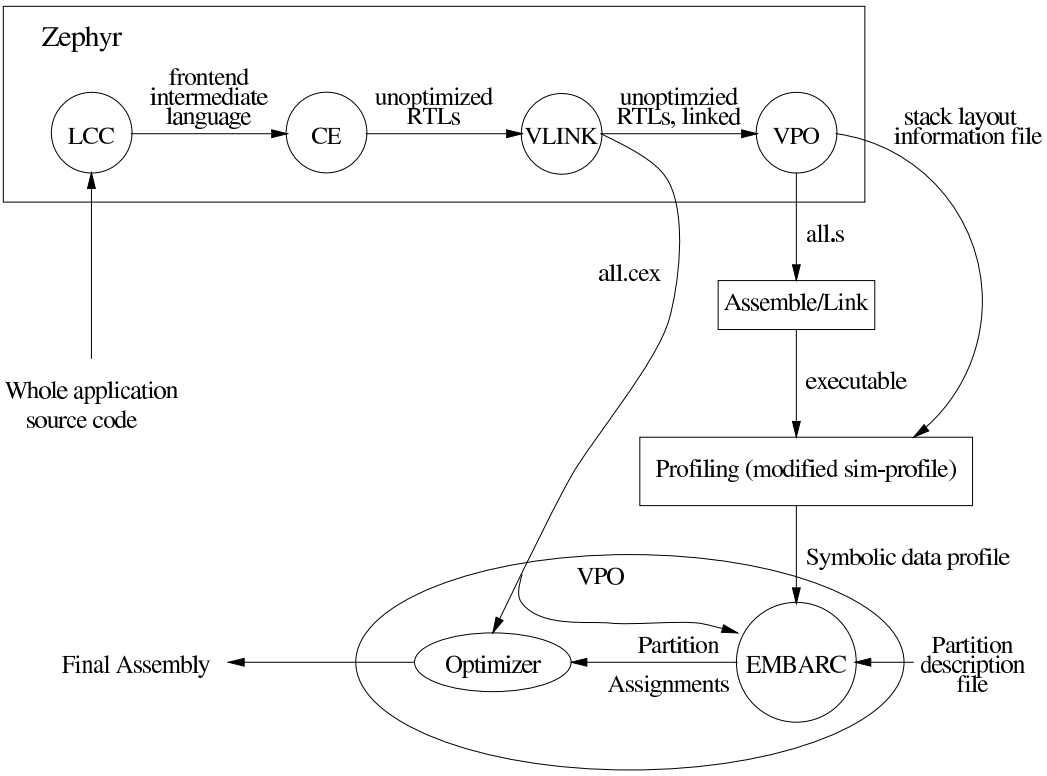

Figure 3.2: Diagram of how EMBARC fits within Zephyr

to implement the algorithm to recompile the entire program with the partition decisions. Although mostly for ease of implementation, there are additional benefits to this approach. Some partition assignments may cause more or less complicated address generation code which may require further optimization. Code motion, register assignment, and common subexpression elimination could make different (and possibly better) decisions after knowing the final assignment. Because of improvements made in other optimization phases, the function prologues/epilogues may be simpler also.

Now that the overall system architecture has been described, the following section discusses one of the central contributions of this research, EMBARC.

3.2 EMBARC

EMBARC, or **E**fficient **M**emory **B**ank **A**ssignment algorithm for **R**etargetable **C**ompilers, is an algorithm to assign variables to memory partitions. EMBARC is a retargetable algorithm in the sense that the memory hierarchy is not fixed *a priori*. Instead, EMBARC reads a description of the memory hierarchy.

After reading the memory hierarchy description, EMBARC reads a dynamic profile. The profile is generated by a profile run of the program and consists of a set of "pseudo-live ranges." Each live range consists of a variable name, a start and end cycle that the variable is live, and a count of how many times the variable was accessed during the live range.

Once the input files are read, EMBARC estimates the access time of each component in the memory hierarchy. EMBARC next calculates the cost of placing each variable into every memory partition. Variables are considered in order from most used to least used, and placement cost is determined based on the variable's access frequency, the partition's estimated access times, and the amount of cache conflict the variable creates when assigned to the partition. The variable is then assigned to the partition that is least expensive for the variable.

The remainder of this section is organized as follows: Section 3.2.1 describes the lan-

guage for describing the memory hierarchy, while Section 3.2.2 discusses the dynamic profiling technique. Finally, Section 3.2.3 explains the partition assignment algorithm which uses the partition description input and profiling input.

3.2.1 Partition Description Language

One input to EMBARC is a description of the possible memory hierarchies on the target machine. We choose to use a description of the memory hierarchy as an input to EMBARC so that EMBARC would be easily retargetable and to allow the end user to experiment with different memory hierarchies. The partition description could be accomplished in other ways, such as implementing C functions to describe the memory partitions or incorporate in the compiler functions to infer the memory configuration of the machine. However, these methods require that the compiler be rebuilt with any change to the partition description or to the memory hierarchy to the current machine. Being able to quickly and easily retarget the memory hierarchy without rebuilding the compiler is a key advantage for fast and easy experimentation. Enabling such experimentation was one of the goals of this research.

A partition description lists each partition and its key access characteristics for each memory hierarchy. It also contains information about the caches, and how the caches link together to form the different memory hierarchies. EMBARC uses information contained in a description of the memory hierarchy to estimate the cost of placing each variable in a given partition. Table 3.1 gives a grammar for the language. In the table, terminal symbols are given in uppercase and non-terminals are enclosed in angle brackets. Figure 3.3 contains a sample description of a memory hierarchy.

A partition description is made up of *partition* and *cache* blocks (productions 2-3 in Table 3.1) which describe memory partitions and caches, respectively. A block is started by specifying the type of block (cache or partition) and the name of that cache or partition. Blocks are terminated by the “end cache” or “end partition” keywords. Each block contains a list of attributes for the entity named at the start of the block.

Inside a partition block, the user specifies the access attributes for the specified par-

```
(1)          <pdlist> ::=  <pd> | <pd> <pdlist>
(2)             <pd> ::=  <parts_list> <cache_list>
(3)      <parts_list> ::=  <part> <parts_list> | <part>
(4)      <cache_list> ::=  <cache> <cache_list>
(5)           <part> ::=   PARTITION  IDENTIFER
                           <part_items>
                           END  PARTITION
(6)      <part_items> ::=  <part_item_list>
                           | <part_item_list>  OR <part_items>
(7)  <part_item_list> ::=  <part_item> <part_item_list> | <part_item>
(8)       <part_item> ::=   SIZE '=' <cocl>
                           |  ACCESS_LATENCY '=' <cocl>
                           |  DEFAULT
                           |  OFFCHIP
                           |  PORTS '=' <cocl>
(9)          <cache> ::=   CACHE  IDENTIFIER <cache_items>  END CACHE
(10)     <cache_items> ::=  <cache_item_list>
                           | <cache_item_list>  OR <cache_items>
(11) <cache_item_list> ::=  <cache_item> <cache_item_list>
                           | <cache_item>
(12)      <cache_item> ::=   SETS '=' <cocl>
                           |  BLOCKSIZE '=' <cocl>
                           |  ASSOC '=' <cocl>
                           |  REPLACEMENT '='  IDENTIFIER
                           |  HIT_LATENCY '=' <cocl>
                           |  EXTRA_MISS_LATENCY '=' <cocl>
                           |  PARTS_CACHED '=' <ident_list>
                           |  MISSES_SERVICED_BY '='  IDENTIFIER
                           |  OFFCHIP
                           |  PORTS '=' <cocl>
(13)      <ident_list> ::=   IDENTIFIER ',' <ident_list> |  IDENTIFIER
(14)          <ident> ::=   IDENTIFIER
(15)           <cocl> ::=   CONSTANT | '{' <con_list> '}'
(16)        <con_list> ::=   CONSTANT |  CONSTANT ',' <con_list>
```

Table 3.1: BNF form of Partition Description Language

tition (production 8). The access latency of a partition can be specified by the **ACCESS_LATENCY**=<*cocl*> attribute. For simplicity, access latency specifies the time to access memory in the common case. The **PORTS**=<*cocl*> attribute specifies how many simultaneous accesses the memory bank can support. The **DEFAULT** attribute, if present specifies that the partition is the default partition, and any variable can legally be assigned to this partition. The *cached_dram* partition in Figure 3.3 takes 10 cycles to access, can satisfy one request at a time, and is the default partition for this description. It is necessary to specify the default partition for at least one partition in the description, and illegal to specify more than one default partition. The default attribute is necessary for the compiler to specify which partition to place variables that cannot be assigned to an arbitrary partition because of missing alias analysis information. The default partition is assumed to be large enough for all variables. The **SIZE**=<*cocl*> attribute specifies the size of the partition. It is necessary to specify the size for all partitions, except the default partition. If EMBARC is considering power in its cost metrics, the size of the default partition should be specified. The order of partition blocks does not matter.

Inside a cache block, the user specifies the access characteristics for the named cache (production 12). The **SETS**=<*cocl*>, **BLOCKSIZE**=<*cocl*>, and **ASSOC**=<*cocl*> specify the number of sets in the cache, the blocksize of each line in the cache, and the associativity of the cache, respectively. The **REPLACEMENT**=$IDENTIFER$ specifies the replacement policy for the cache, legal values are l for LRU, f for FIFO, and r for pseudo-random. The **HIT_LATENCY**=<*cocl*> and **EXTRA_MISS_LATENCY**=<*cocl*> are used as the hit latency, and time to detect a miss[2], respectively. Each cache must also have a **PARTS_CACHED**=<*ident_list*> attribute to specify which partitions are cached by this cache. If a cache has a lower-level cache backing it, the partition description needs a **MISSES_SERVICE_BY**=**IDENTIFER** attribute to specify which cache handles misses in this cache. In the Figure 3.3, the *dl1* cache is set to 256 sets, each containing 1, 32-byte

[2]Note that the **EXTRA_MISS_TIME** attribute does not specify the time to satisfy a miss. The total miss time is calculated by the time to access lower levels of cache or main memory and the **EXTRA_MISS_TIME**.

line. It takes one cycle to access the cache, and cache misses are serviced by the *ul2* cache.

Most attributes support not just a single value, but a list of values (via productions 15-16). If a list of values is used anywhere in the description, the description is describing multiple memory hierarchies. For example, in Figure 3.3, the level-2 cache, *ul2*, is specified as 1-, 2-, or 4-way associative. Specifying a list of values is useful if the user wishes to create partition assignments for multiple memory hierarchies so that each assignment can be evaluated. Embedded processor designers may need to evaluate many memory hierarchies for a proposed chip, so describing many memory hierarchies is a useful feature of EMBARC's partition description language.

The partition description language does not provide a way to describe more complicated structures such as store buffers or prefetching mechanisms. Although these features may enhance the hardware performance, in the interest of simplicity and ease of use, these features are not included in the language. Thus, the language contains only information which is absolutely essential for EMBARC to generate effective partition assignments. Such simplicity also provides ease of use for the end user. More advanced features are very complicated to describe and work differently from machine to machine.

The simplicity of the language not only helps the user read and write the partition descriptions, it makes the compiler-writer's job easier too. Since the compiler writer must implement a parser for the language, a simple language promotes widespread usage of the language. Our implementation of a parser is only 155 lines of *yacc* code, supported by 55 lines of *flex* code for token scanning and about 800 lines of C to build a simple data structure.

3.2.2 Profiling

To get an accurate view of the data access patterns of the program, EMBARC requires a dynamic data profile using a sample input (see Figure 3.2). The dynamic profiling technique was chosen because it has complete and accurate information about the memory accesses during the execution of the program, at least for the sample input. In contrast, a technique

```
partition cached_DRAM
        access_latency=10
        ports=1
        size=10485760 # 10 megabytes
        default  # default partition marker
end partition
cache il1 # 8k i-cache
        sets=128
        blocksize=64
        assoc=1
        replacement=l
        hit_latency=1
        extra_miss_latency=1
        parts_cached=cached_DRAM
        misses_serviced_by=ul2
end cache
cache dl1 # specify the primary d-cache
        sets=256
        blocksize=32
        assoc=1
        replacement=l  # LRU replacement
        hit_latency=1
        extra_miss_latency=1
        parts_cached=cached_DRAM
        misses_serviced_by=ul2
end cache
cache ul2 # 64k or 32k L2 cache
        sets={1024,2048}
        blocksize=64
        # 1, 2 or 4 way assoc
        assoc={1,2,4}
        replacement=l
        hit_latency=10
        extra_miss_latency=10
        parts_cached=instructions,cached_DRAM
end cache
```

Figure 3.3: A sample memory hierarchy description

which uses only information about the static program text may be woefully incomplete or inaccurate in some cases (particularly for dynamic allocation, and pointer intensive codes). To make EMBARC as general as possible, the dynamic profile technique is used.

The dynamic profile contains information about which variables are accessed during which portion of the program. In particular, for each variable in the program, a list of "live ranges" along with reference counts is maintained. In order to keep a variable from being live when in fact it is not used, a cutoff value is used. If a variable has not been accessed for X cycles, it is then considered dead during those cycles. The smaller the value of X the more fine-grained the profiling becomes. If X is 1, that is equivalent to having a complete trace of memory accesses. Other dynamic profiling techniques (such as recording basic blocks sequences or hot traces) may have been appropriate, but this technique allows the user to tune of the size of the profile. It also allows EMBARC to exploit the program's data accesses patterns and calculate how frequently variables might conflict in a cache if mapped to the same cache lines.

For the experiments described in Chapter 5, the cutoff value was set to 1,000 cycles because that supplied EMBARC with sufficient information about the benchmarks, without creating profiles that were too large to store. Typically, the above technique generated profiles under 1 megabyte, while on one benchmark, *dijkstra*, the profile was approximately 200 megabytes.

To produce dynamic profile information, a simulator (based on SimpleScalar) that monitored each load and store instruction to a particular variable is used [14]. The simulator maintains a hash table that contains each variable, and each variable contains a sorted list of live ranges. When an address is read or written, the simulator determines which variable is accessed and updates the appropriate live range list.

To determine the addresses that are bound to a variable, the simulator keeps a list of which variables are bound to which addresses. The list is initialized with global variable start and end addresses gleaned from the linked executable. To track heap variables, the simulator watches for calls to the C runtime libraries *malloc* and $free$ and updates the

list appropriately. Lastly, the compiler provides information about the stack layout which is used by the simulator to bring local variables into and out of scope during call/return instructions. To efficiently search this list, the simulator maintains a move-to-front heuristic and relies on data-symbol locality. We have found this approach to work very efficiently for the benchmarks[3] used in this research and the simulator searches on average 10 elements of the list.

Unfortunately, memory accesses are not always traceable to program variables. Calls to library routines and dynamically allocated stack variables (using *calloc*) can prevent the simulator from knowing information about which variable is accessed in a load or store. However, for the benchmarks used in this research, over 99% of all memory accesses can be traced to variables using the techniques described above. Thus, these techniques provide a thorough dynamic data profile that tracks global, local, and heap variables.

The simulator's runtime is sufficiently fast for the purposes of this research (profile simulations took less than 25 minutes per benchmark to complete.) However, if the runtime overhead of simulation is too expensive, these same techniques could be applied to a native binary by having the compiler (or binary editor) instrument the binary. Each load and store would need a call to an accounting routine, while each function prologue/epilogue would need a similar call. Also, a customized *malloc* and $free$ would be required to signal the runtime system when new dynamically allocated variables enter and leave scope.

3.2.3 Partition Assignment

EMBARC uses the information in a partition description file, and the output from Zephyr and SimpleScalar to generate a partition assignment for each variable in the program. First, EMBARC calculates an "estimated average access time" ($EAAT$) for each cache in the partition description. For caches, a 90% hit rate is assumed. The $EAAT$ for a cache,

[3]See Chapter 5 for a discussion of the benchmarks used for evaluation.

C, with a backing cache, BC, cache is:

$$C_{EAAT} = .9 * T_{hit}(C) + .1 * (T_{miss}(C) + BC_{EAAT})$$

While the $EAAT$ for a cache, C, backed by memory from partition, P, is:

$$C_{EAAT} = .9 * T_{hit}(C) + .1 * (T_{miss}(C) + T_{access}(P))$$

where $T_{hit}(C)$ and $T_{miss}(C)$ are the hit and miss times for cache C and $T_{access}(P)$ is the time to access partition P. This is a straightforward calculation of an cache average access time assuming a 90% probability of a hit from a backing cache, or backing partition. Since partition assignments have not been made, it is impossible to know which variables are accessed via which cache. Consequently, a more accurate estimation of the cache hit rate is unavailable at this point. Since the estimates are only used for partition assignment, it is not important that they are totally accurate. It would be possible to make a better estimate once the partition assignment is complete, and then iterate the assignment using the new estimates. However, we found this to be unnecessary because most caches are well behaved and the algorithm performed well.

Finally, the $EAAT$ for a partition, P, is calculated. To calculate P_{EAAT}, the estimated access time for each level-1 cache that caches the partition is averaged. If there are n top-level caches that cache partition P, C^1 to C^n, the P_{EAAT} is:

$$P_{EAAT} = \frac{\sum_{i=1}^{n} C^i_{EAAT}}{n}$$

This calculation assumes that each cache is accessed an equal number of times. Like calculation for the cache hit rate estimates, partition assignments are unavailable at this point. Because of the unknown partition assignments, a more accurate estimation of cache access frequencies is unavailable during this phase.

In the case where a partition has no caches associated with it (n is 0 in the previous

equation), then P_{EAAT} is simply the access time for P. Although having uncached partitions may seem unlikely, many embedded systems are cacheless or include an uncached on-chip SRAM partition, so the $n = 0$ case occurs often in practice.

After calculating $EAAT$ for each memory component, EMBARC assigns a partition to each variable, v, in the program. It considers variables in decreasing order of number of accesses. To determine which memory bank is the best assignment for the variable, the algorithm computes the cost of placing the variable in each bank. The cost for placing a variable, v, in a partition, P, is:

$$cost_v^P = tcf(v, P)^2 + (P_{EAAT} * v_{refs})^2$$

Where $tcf(v, P)$ is defined as the total conflict factor between variable v and all other variables already assigned to partition P (see Section 3.2.3.1). The idea captured by this equation is that the cost is directly related to the number of references and the time it takes to access the variable in the given partition. Furthermore, the cost increases if the variable has a high conflict factor in the partition.[4]

3.2.3.1 Total Conflict Factor Calculation

The tcf represents how many cache conflicts a variable would have with another variable or when assigned to a new partition. We use a similar definition of tcf as Panda, but ours comes from the dynamic data profile instead of the static program text [41,40]. We use our own definition of tcf because the static program text is often difficult to analyze on larger programs. On the other hand, the dynamic profile is accurate even on very large, pointer intensive applications with extensive dynamic memory allocation. The drawback to using a dynamic profile is that the algorithm may be highly sensitive to the particular data used for profiling the program. Section 5.1.7 discusses profile input sensitivity in more detail.

To calculate the dynamic tcf for two variables, v_1 and v_2, which are being considered

[4]Note that the $tcf(v, P)$ is zero if partition P is uncached.

for assignment to the same partition, the tcf of each overlapping live range is examined. The live range for each variable is a triple, (s, e, n), which starts at cycle s, ends at cycle e and has n accesses. If two live ranges, $r_1 = (s_{v_1}, e_{v_1}, n_{v_1})$ and $r_2 = (s_{v_2}, e_{v_2}, n_{v_2})$ overlap, then the percentage overlap of each access is calculated. If r_2 overlaps o_{r_1,r_2} percent of r_1, while r_1 overlaps o_{r_2,r_1} percent of r_2, the tcf of r_1 and r_2 is:

$$tcf(r_1, r_2) = \frac{n_{v_1} * o_{r_1,r_2} + n_{v_2} * o_{r_2,r_1}}{2}$$

To calculate the tcf for v_1 and v_2, the tcf of each pair of ranges for v_1 and v_2 is summed. To calculate the $tcf(v_1, P)$, the tcf between v_1 and each variable already assigned to partition P is summed.

Although this sounds rather computationally expensive, it is a simple process if the live ranges are maintained in sorted order by starting cycle. Since the profiler generates the live ranges in order, no explicit sorting is necessary. Sorting this way simplifies the tcf calculation between two variables since it is not necessary to calculate a tcf between two non-overlapping ranges. Observing this, it is straightforward to calculate $tcf(v_1, v_2)$ in $O(n_{v_1} + n_{v_2})$. It may still seem computationally expensive to calculate a tcf between v and every variable assigned to a partition P. However, once an assignment to a partition has been made, the live ranges for that variable can be merged into a single list of live ranges for partition P.

Even though our implementation does not make these simplifications, we found that the runtime of the tcf calculations to be very reasonable. It never took more than several minutes to calculate partition assignments, even when there were thousands of dynamically allocated variables to consider.

Once the cost of placing v in each partition is determined, the algorithm simply chooses the partition with the lowest cost. In the case of a tie (when few variables have been assigned to otherwise equal cost partitions or when variables are not seen in the dynamic profile) the static program text is used to determine a second tcf, as in Panda's work, to

break the tie [41, 40].

Chapter 4

Estimating Memory Hierarchy Performance

As was discussed in Chapter 2, an embedded system designer needs to quickly evaluate many possible memory hierarchies. EMBARC and its ability to effectively assign variables to multiple memory partitions is important when evaluating different memory hierarchies, but it is also important to know how efficiently the memory hierarchy satisfies the requests to access these variables. Together, the ability to assign variables and to estimate the performance of a memory hierarchy allow an embedded system designer to quickly choose a memory hierarchy that best meets the system's needs.

This chapter discusses MPRES, an algorithm to estimate the performance of a memory hierarchy on a given application. Figure 4.1 contains a diagram illustrating how EMBARC and MPRES work together within VPO to make performance estimates for a candidate memory hierarchy. The figure shows that MPRES takes the description of the memory hierarchy, the dynamic profile, and the partition assignments generated by EMBARC, and emits an estimate of the time needed to satisfy requests from the memory hierarchy.

Figure 4.2 contains pseudo-code for the MPRES algorithm. As the pseudo-code shows, MPRES first estimates the hit rate for each cache in a top-down fashion (i.e. starting with caches closer to the processor core). MPRES then estimates the average access time for each cache and partition. Finially, MPRES estimates the amount of time the processor will spend satisfying memory requests. The following sections describe these steps in more

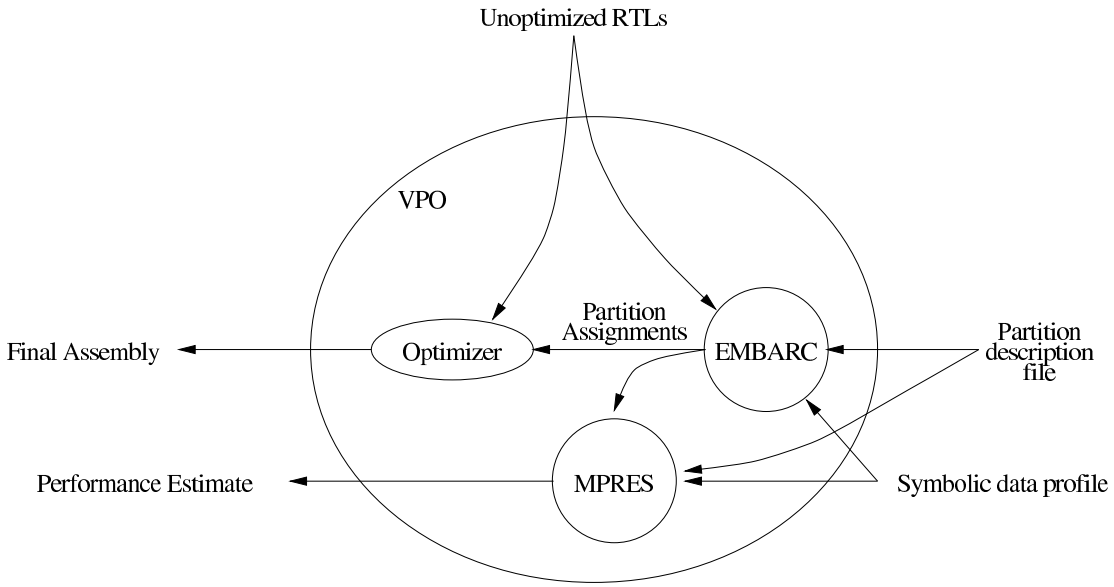

Figure 4.1: Diagram of VPO with EMBARC and MPRES

detail.

4.1 Estimating Cache Hit Rates

Once partition assignments are made, the data profile can be used to estimate the access patterns to each cache. In this work, the cache hit rate for each cache in the memory hierarchy is calculated first. To do this, the estimation phase calculates an *effective cache size* for each cache, C, in a top-down fashion, as follows:

$$C_{\text{effective\_size}} = C_{\text{size}} * (1 + (1 - \frac{\sum_{\forall v_1, v_2 \in P_C} tcf(v_1, v_2)}{C_{\text{assoc}} * C_{\text{accesses}}}))$$

where P_C is the partition cached by C. The summation is divided by the cache's associativity, C_{assoc}, to account for the reduced conflicts from higher associativity. The sum is also divided by the accesses to cache C. This division is done to convert the summation into a percentage, representing the percentage of conflict.

Next, using the data profile sorted by access count, the estimation phase counts the accesses to variables until the sum of the variables' size exceeds the effective cache size.

```
double mpres()
{
        for each cache, C, in a top-down fashion
                calculate C_effective_size
                calculate S, the number of hits for C
                calculate C's hitrate, C_hr
        end for

        for each cache, C, in a bottom-up fashion
                calculate C_EAAT
        end for

        for each partition, P
                calculate P_EAAT
        end for

        sum=0.0
        for each partition, P
                sum = sum + P_accesses * P_EAAT
        end for

        return sum
}
```

Figure 4.2: Pseudo-code for MPRES

Figure 4.3 shows an example of how this is done. In the figure, the accesses to variables 1, 2 and 3 are included in the summation. Also, part of the accesses to variable 4 are included.[1] Call this sum S. The estimate for C's hit rate is defined as:

$$C_{hr} = \frac{S}{C_{ls} * C_{\text{accesses}}} + 1 - \frac{1}{C_{ls}}$$

where C_{ls} is the C's line size, and C_{accesses} is the number of accesses to C. The idea is that most of the S accesses will be hits and most of the remaining accesses will be misses. However, we also want to take into account that while traversing an array, which is common in embedded codes, about $1/C_{ls}$ accesses will probably be misses.

Once C_{hr} is calculated, the estimation of misses can be passed down to lower level caches as cache accesses, and the calculation is repeated (omitting the variables believed to be cached by the upper level caches, such as Var 1-3 in the example) to calculate hit rates for lower level caches.

4.2 Estimating Average Access Times

Once all the caches have an estimated hit rate, we estimate the average access time for each cache. Since the access time of a first level cache depends on access times for second level caches, we do this in a bottom-up fashion. Each cache is estimated to take time:

$$C_{EAAT} = t_C * C_{hr} + t_m * (1 - C_{hr})$$

where t_C is the time to access C, and t_m is the time to to satisfy a miss (either by using the $EAAT$ of a lower level cache, or by the time of the backing partition.) This is the same technique used for making the initial $EAAT$s during the partitioning phase, but a better estimate for the cache hit rate is used to calculate the C_{EAAT}

[1] The estimation phase calculates the percent of the variable's size that fits within the cache, say $X\%$. It then includes $X\%$ of the accesses in the sum.

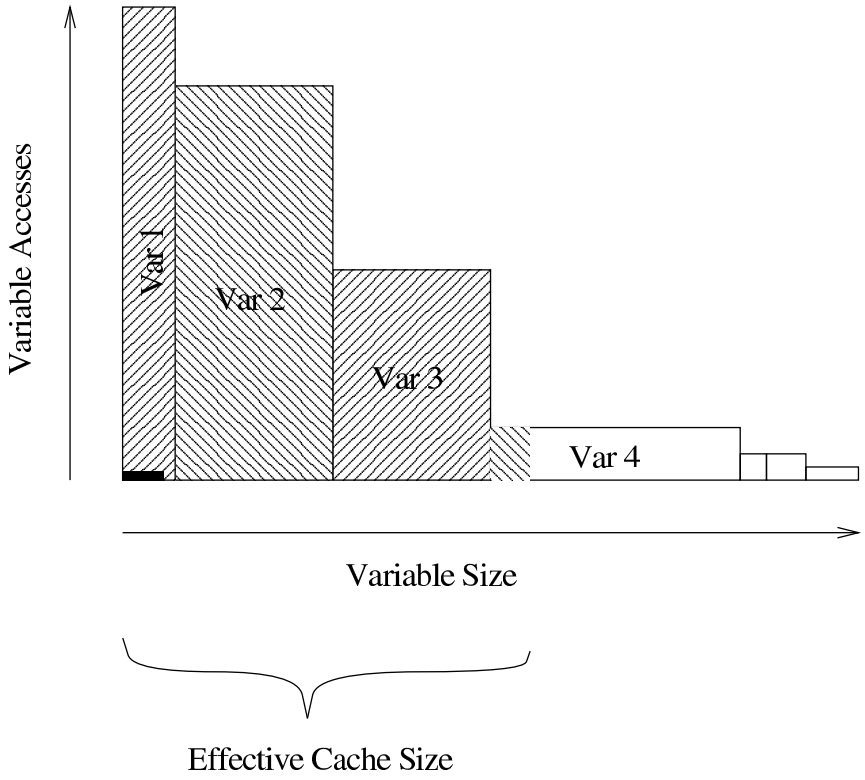

Figure 4.3: Visualization of cache hit rate calculation.

Finally, once each cache has been assigned an $EAAT$, the average access time for each partition can be calculated by seeing which caches are used for each partition. Lastly, the total access time is calculated thusly:

$$\text{total access time} = \sum_{P \in \text{partitions}} P_{\text{accesses}} * P_{eaat}$$

This gives an estimate of how long the processor will spend satisfying memory requests if it were to re-run the same program on the training input with the given memory hierarchy.

Together, the EMBARC and MPRES algorithms allow embedded system designers to experiment efficiently and effectively using vastly different memory hierarchies. However, for these tools to be useful, they must provide effective partition assignments and memory hierarchy performance estimates that closely match the performance of real systems. Chapter 5 evaluates the effectiveness and accurateness of the EMBARC and MPRES algorithms.

Chapter 5

Evaluation

This chapter describes experiments performed to evaluate the effectiveness of the EMBARC and MPRES algorithms. Section 5.1 describes the evaluation of EMBARC, while Section 5.2 describes the evaluation of MPRES.

5.1 Memory Partitioning Quality

The results of the following experiments demonstrate that EMBARC generates partition assignments that are as effective as partition assignments from memory-hierarchy specific algorithms. Section 5.1.1 describes the experimental setup used for the evaluations, while Sections 5.1.2–5.1.4 compare EMBARC to memory-hierarchy specific algorithms. Sections 5.1.5 and 5.1.6 demonstrates EMBARC's flexibility by discussing results from a system with 2 data caches and using a cache bypassing mechanism, respectively. Section 5.1.7 discusses EMBARC's sensitivity to the profiling inputs used for creating the dynamic data profile.

5.1.1 Experimental Setup

The EMBARC framework as described in Chapter 3 is implemented in VPO and targeted to the SimpleScalar PISA instruction set [14, 11]. PISA is a variant of the MIPS R4000 instruction set, commonly used in some embedded processors. PISA is a RISC instruc-

tion set with most common operations arithmetic instructions supported for registers only. Thirty-two integer registers, and 32 floating-point registers are available, but three of the integer registers are reserved for the stack pointer, frame pointer, and global pointer.

SimpleScalar (with the Wattch extensions) is configured to use a 2-issue out-of-order processor in all experiments [13]. The CPU has two integer execution units, and one floating-point execution unit, along with a memory unit for each partition. All on-chip caches and SRAM are modeled as having a one cycle latency, while the 64K off-chip cache (level-2 cache) is modeled as having a 10 cycle latency. In the experiments with a level-2 cache, DRAM is modeled as having 100 cycle latency, while in the cacheless experiments, a smaller, 10-cycle DRAM is used. We feel the system described above is representative of modern embedded systems.

The Wattch extensions perform a cycle-accurate power and energy simulation of all the major on-chip structures. During each cycle, a fixed energy cost is charged for each structure that is accessed. If a structure is not accessed, 10% of the structure's full access cost is charged for leakage. See [13] for more details on the *CC3* energy model used in the experiments.

5.1.1.1 Benchmarks

To make a fair and meaningful comparison with previous research, a wide variety of benchmarks are used to evaluate the EMBARC algorithm. Tables 5.1 and 5.2 describe the benchmarks that are used to evaluate the algorithms developed, along with information about the data segment. It also lists the previous research that has used these benchmarks along with information about how to obtain these benchmarks (second column). The table also contains information about the program's variables: column three lists the number of static program variables, column four lists the total size of those variables, and column five lists the dynamic instruction count when the program is using its reference input.

Some benchmarks have several variations. *Edge_detect* has two versions, one with inlined functions to avoid alias analysis issues that Saghir must have avoided [44]. The other

is the original version. Several benchmarks have the *.nl* extension, which denote that local variables are manually converted to global variables. Such local to global promotion is performed in some research performed by Avissar [7,8]. We include them for a fair comparison. The number of source variables and total data size for some benchmarks is omitted because of the large text size and dynamic allocation properties of some benchmarks.

Table 5.3 describes the inputs used for profiling and for the reference input. Note that the benchmarks from Panda's research are omitted, as they take no input and are kernel loops only [40].

Some benchmarks used in prior research, unfortunately, are too small to be considered valid data points. *Dequant*, for example, only executes approximately 8,000 instructions. Such benchmarks are included to compare against previous research and to illustrate the importance of appropriate benchmarking in memory partitioning research.

When appropriate, benchmarks have been given larger inputs to ensure the benchmarks are long-running enough to gain accurate measures of the kernels loops and are not measuring start-up or one-time codes.

5.1.2 Cacheless without SRAM

To evaluate the effectiveness of the EMBARC algorithm on systems without caches or SRAM, we compare EMBARC results to Saghir's published results [44]. We perform a set of experiments similar to those in Saghir's research on the same set of benchmarks. In particular, we compare a standard singled-ported memory (*baseline*) against an *ideal* dual-ported memory and a partitioned memory (*2banks*). Saghir's implementation uses static scheduling, however the *edge_detect* benchmark has one function (called with the addresses of global variables) that does most of the processing of the benchmark. The static scheduling mechanisms would need some form of inlining in order to get good speedups for this function. Since Saghir's implementation apparently uses inlining, we include *edge_detect.inline* which has been manually inlined.[1] Figure 5.1 shows the results of this set of experiments. The

[1]The Zephyr infrastructure does not support automatic inlining.

Benchmark	**Source**	**# Src Vars**	**Memory Size (bytes)**	**Instr. Count**	**Description**
Beamformer	[40]	6	16,640	33M	DSP application representing temporal alignment and summation of digitized signals
DHRC	[40]	4	936	24K	Differential heat release computation
Dequant	[40]	5	2,560	8K	Dequantization routine in an MPEG decoder application
FFT	[40]	3	8,100	343K	Standard FFT kernel, no I/O.
IDCT	[40]	3	1,536	530K	Inverse discrete cosine transformation
MatrixMult	[40]	3	3072	50K	Standard Matrix Multiply (16x16)
SOR	[40]	7	280,000	331k	Successive over relaxation
CRC32	[7, 24]	NA	NA	4.8M	Compute the 32-bit CRC used as the frame check sequence in ADCCP
dijkstra	[7, 24]	NA	NA	285M	Dijkstra's shortest path algorithm
fft_kernel	[7, 3]	NA	NA	21M	256-point complex FFT (radix-2, in-place, decimation-in-time)
bmm(.nl)	[7, 2]	5(25)	240,008 (240,108)	110M (174M)	Block Matrix Multiply
btoa(.nl)	[7, 2]	11(17)	1,648 (1,672)	1.1M (1.4M)	Binary to ASCII converter
fib(.nl)	[7]	0(4)	0(16)	6.0M (26M)	Fibonacci number computation

Table 5.1: Description of Benchmarks

Benchmark	**Source**	**# Src Vars**	**Memory Size (bytes)**	**Instr. Count**	**Description**
fir(.nl)	[7]	0(2)	0(8)	3.2M (7.6M)	FIR filter code
adpcm encode	[44, 3]	NA	NA	7.4M	Adaptive, Differential, Pulse-Code Modulation Speech Encoder
adpcm decode	[44, 3]	NA	NA	6.6M	Adaptive, Differential, Pulse-Code Modulation Speech Decoder
G721ML encode	[44, 3]	NA	NA	108M	Implementation of CCITT G.731 ADPCM Speech Encoder
G721ML decode	[44, 3]	NA	NA	74M	Implementation of CCITT G.731 ADPCM Speech Decoder
lpc	[44, 3]	NA	NA	67M	Linear Predictive Coding speech encoder
trellis	[44, 3]	NA	NA	30M	Trellis decoder
edge_detect(.inline)	[44, 3]	NA	NA	150M	Edge Detection using 2D convolution and Sobel operators
pegwit.decode	[24]	NA	NA	89M	Public key decryption and authentication
mpeg2.decode	[24]	NA	NA	159M	Converts a compressed bitstream into an ordered set of uncompressed output pictures

Table 5.2: Description of Benchmarks (cont.)

Benchmark	**Profile Input**	**Reference Input**
CRC32	*small.pcm* – 1,368,864 byte pcm file	*large.pcm* – a 26,611,200 byte pcm file
dijkstra	*input.dat 20* – 20 shortest path searches	*input.dat 50* – 50 shortest path searches
fft_kernel	no input, kernel only	no input, kernel only
bmm(.nl)	multiply 100x100 array in 50x50 chunks	multiply 100x100 array in 50x50 chunks
btoa(.nl)	*btoa.man* – manual for btoa benchmark	*long.btoa.man* – 6 copies of btoa.man
fib(.nl)	Finds first 1,000 Fibonacci numbers	Finds first 1,000,000 Fibonacci numbers
fir(.nl)	no input, kernel only	no input, kernel only
adpcm.encode	*clinton.pcm* 295,040 byte pcm file	*clinton.pcm*
adpcm.decode	*clinton.pcm*	*clinton.pcm*
G721ML.encode	*bark.pcm* – 2,407 sample pcm file	*bark.pcm*
G721ML.decode	*bark.pcm*	*bark.pcm*
lpc	73k speech sample, processed 40 times	73k speech sample, processed 400 times
trellis	randomly generated input array	randomly generated input array
edge_detect(.inline)	2 iterations on 53k picture	20 iterations on 53k picture
pegwit	91k pgptest.plain (encoded)	637k pgpref.plain (encoded)
mpeg2.decode	*test.m2v* – 8k m2v file	*mei16v2.m2v* 35k m2v file

Table 5.3: Description of Benchmark Inputs

bars labeled *ideal* refer to the baseline machine with a single memory bank, but having 2 ports (i.e. any two memory accesses can occur in parallel.) EMBARC is not needed for *ideal* (nor for the *baseline* version), however the *2banks* bars represent when EMBARC is used to partition variables between two equal sized, equal latency (10 cycle) partitions. All bars are normalized to the baseline version (1 memory port).

Most benchmarks get near ideal speedups. However, a few do not; namely *fft_kernel*, *lpc*, and *edge_detect*. Although Saghir reports equal-to-ideal speedups for *fft_kernel*, we find that since our machine model is slightly different (dynamically scheduled instead of statically scheduled) our ideal model actually does better than Saghir's ideal model. The three memory operands per loop iteration takes 20 cycles (two 10 cycle accesses in parallel, followed by a single 10 cycle access) in a statically scheduled machine. However, in a dynamically scheduled ideal machine, the second access can be overlapped with the first access from the next iteration. Thus, two iterations can be completed in 30 cycles instead of 40 cycles for the static scheduled. The solution provided by EMBARC has been verified to be optimal and the two bank system simply cannot perform as well as the ideal system in this case.

Saghir also reports that *lpc* can get no speedup without data duplication (which is not modeled in this work). However, our results show some improvements. We suspect this is also because of slight differences in the machine model. However, we believe that data duplication is clearly needed to get near ideal performance for *lpc*. Lastly, *edge_detect*'s poor performance can be explained by the lack of inlining. Performance is greatly improved when the inlined version is used.

To compare against Saghir's published work, we examine the benchmarks we have in common, namely *fft_kernel, lpc, adpcm, edge_detect.inline, G731ML*, and *trellis*. We exclude *fft_kernel* because both algorithms achieve the optimal partition assignment. For the remainder of the common benchmarks, we see that EMBARC achieves 41.8% of ideal. Saghir reports improvement at 42.4% of ideal. Thus, EMBARC achieves 99% of the performance generated by a dedicated solution. Consequently, we assert that for multibank cacheless

systems, EMBARC can achieve performance gains similar to a dedicated algorithm.

The graph in Figure 5.2 displays the system energy used for the benchmark execution. The trends closely match that of the speedup graph; the faster the program executes, the less energy is needed to execute the program. However, the *2bank* approach typically uses proportionally less energy since it takes less energy to access the smaller partitions.

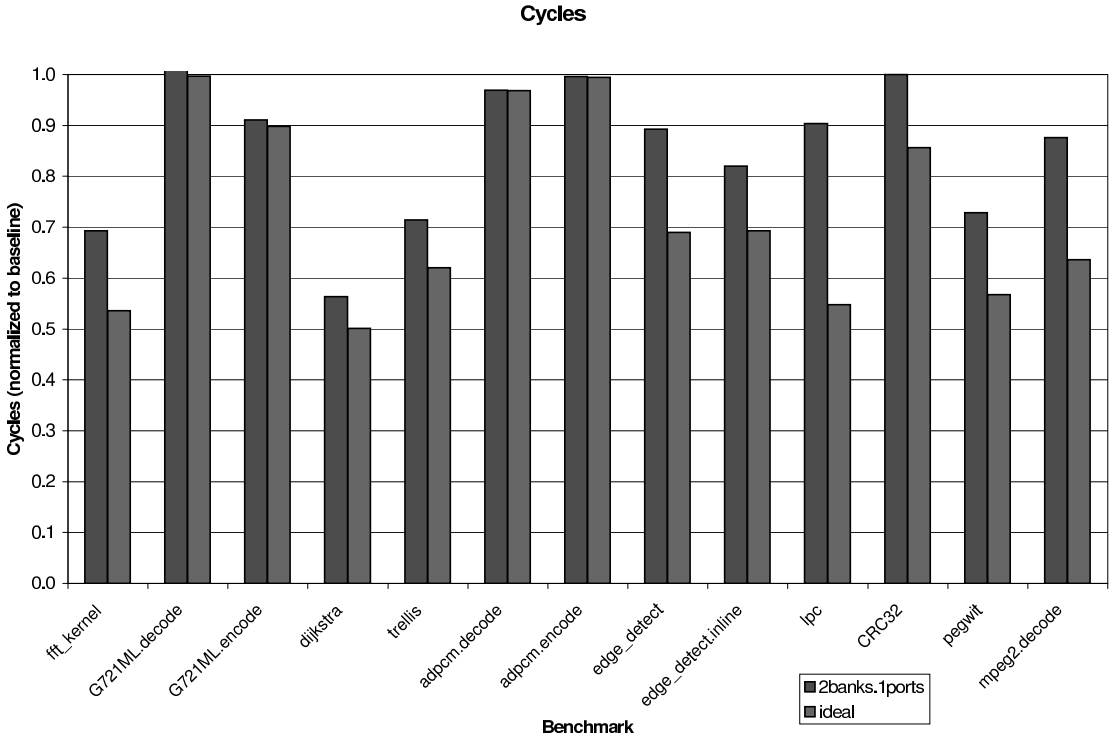

Figure 5.1: Runtimes for systems with 1- or 2-ported DRAMS

5.1.3 Cacheless with SRAM

Avissar demonstrated a method for assigning variables to off-chip DRAM and on-chip SRAM in an optimal manner using integer linear programming [7]. Avissar evaluated his approach on a set of small benchmarks. However, in some of Avissar's experiments, local variables are manually converted to global variables. This change in the program makes the use of SRAM seem extremely beneficial because many loop induction variables are assigned to the SRAM (instead of DRAM) when the original source code would have likely had these variables promoted to registers. To enable comparison of this work to Avissar's published

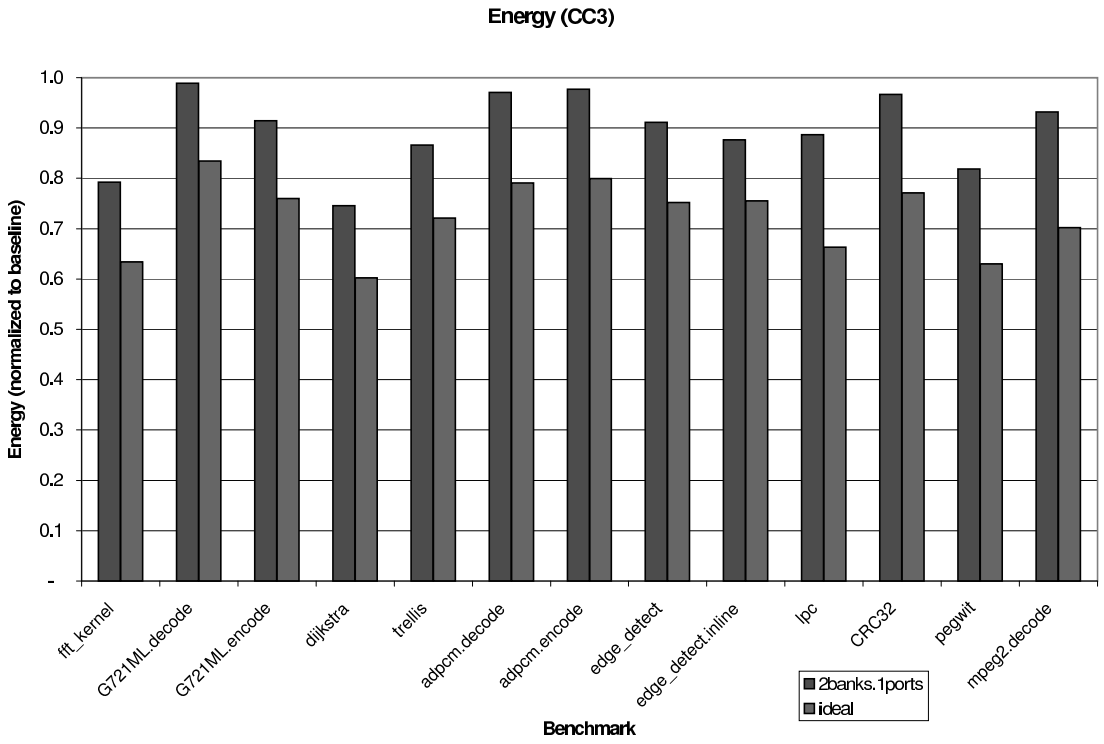

Figure 5.2: Energy for systems with 1- or 2-ported DRAMS

results, we include a set of benchmarks (denoted by the *.nl* suffix) in Figures 5.3 and 5.4 which, like Avissar's research, have undergone the local to global conversion [7].

Figure 5.3 gives the results of an experiment similar to the one performed by Avissar [7]. In that experiment 20% of the program's data segment was allocated as fast (1-cycle) SRAM, while 80% was allocated as slower (10-cycle) DRAM. The partitioning algorithm chooses which variables to place in the SRAM. The bars labeled *20%* show the result of this experiment. The bars labeled *opt20%* show the hand calculated optimal values for these benchmarks (again with 20% SRAM). Note that not all benchmarks have an *opt20%* bar. However, for all the benchmarks simple enough to hand generate an optimal solution (*bmm, fft_kernel, fir, edge_detect, fib, btoa, bmm.nl, fir.nl, btoa.nl,* and *fir.nl*) the figures show that the EMBARC algorithm generates optimal partitioning assignments.

Also included, for completeness, are bars showing how the system performs when 5%-80% of the system is implemented as SRAM. All bars in Figure 5.3 are normalized to the speed of the system when no SRAM is used.

Figures 5.3 and 5.4 show the benefits of a small amount of SRAM. Many benchmarks get significant improvements with as little as 5% of the program's data in SRAM. This fact is particularly true for the *.nl* programs. Loop induction variables are obvious candidates for fast memory.

Figures 5.3 and 5.4 also show that EMBARC generates optimal partition assignments for many simple programs where an ILP solver may be sufficiently fast enough. On the more complex benchmarks (*G721ML, dijkstra, adpcm, lpc, CRC32, pegwit*, and *mpeg2.decode*), the figures show that a small amount of SRAM provides much of the performance improvement that can be gained. For example, when only 5% of the system memory is SRAM, the *dijkstra* benchmark achieves performance that is near the performance of a system using 80% SRAM. When the SRAM is being used effectively, small amounts of SRAM provide the greatest benefits because the most profitable variables are promoted to the SRAM first. Since the figures show this pattern, we conclude that the SRAM is being effectively used. Consequently, these results demonstrate that EMBARC can effectively partition variables in a system with SRAM and DRAM.

5.1.4 Caches and SRAM

To evaluate the EMBARC algorithm on systems with cache and SRAM, two sets of experiments were performed. In both sets, the effectiveness of using 2k of on-chip resources and how that should be allocated was compared: 2k of on-chip cache, 2k of on-chip SRAM, or a hybrid approach with 1k of cache, and 1k of SRAM. The first set of experiments uses the benchmarks that Panda used to evaluate his algorithm (see Table 5.1) [40].

Figure 5.5 shows the execution cycles for each configuration, while Figure 5.6 shows the energy consumption for each configuration. In both figures, the first bar shows the performance if 1k of cache combined with 1k of SRAM when EMBARC chooses the partition assignments. The second bar shows the performance of a 2k cache. All bars are normalized to 2k of on-chip SRAM. As the figures show, the 1k of cache combined with 1k of SRAM performs very similar to the pure cache configuration in four of seven cases (*Beamformer,*

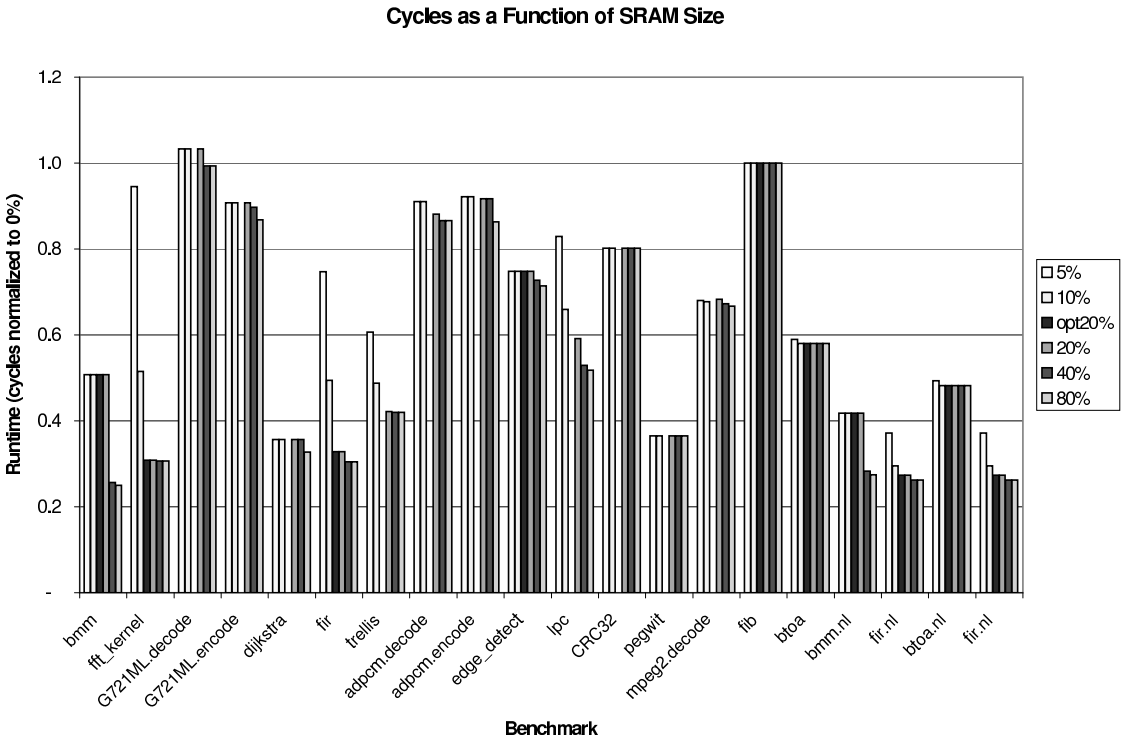

Figure 5.3: Cycle count for systems with 0–80% SRAM

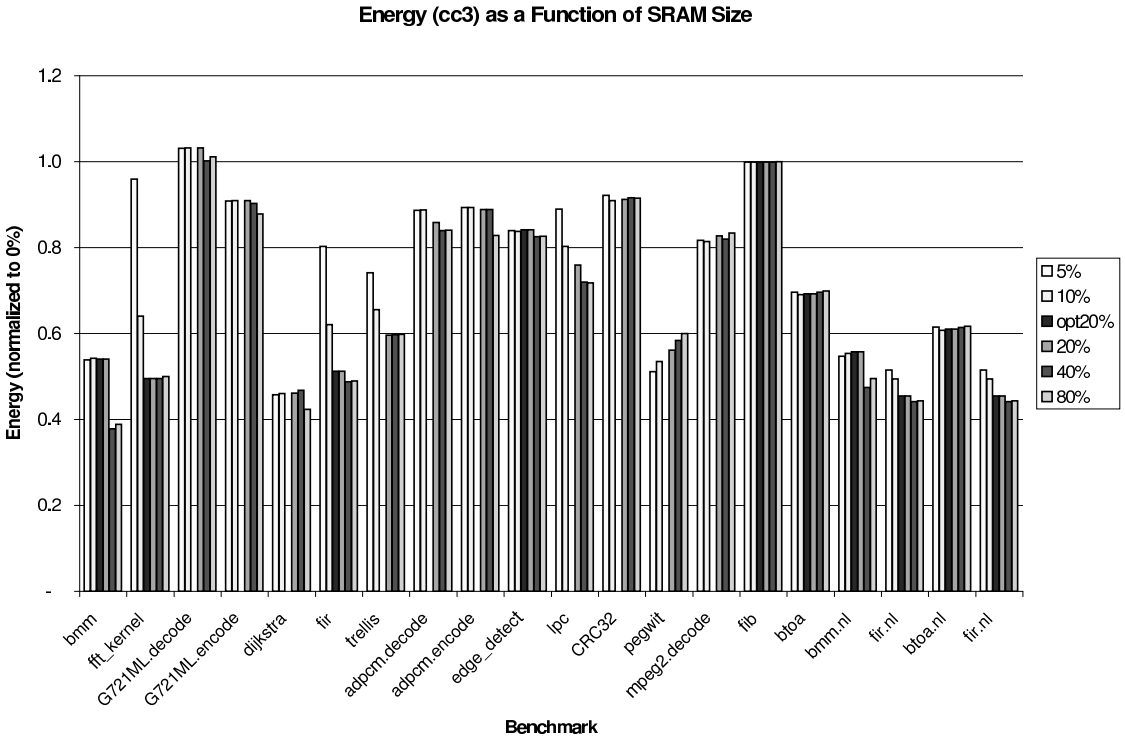

Figure 5.4: Energy count for system with 0–80% SRAM

Dequant, IDCT, and *SOR*). In the remaining cases, it is not clear which memory hierarchy is best overall. For the *DHRC* and *MatrixMult* benchmarks, the mixed SRAM and cache configuration wins, but in *FFT* the pure cache configuration wins. Panda's published research showed a similar pattern [40].

Unfortunately, these figures do not reveal whether EMBARC is generating effective partition assignments. To see EMBARC's effectiveness on these benchmarks, consider Figure 5.7. The first bar is the same as the first bar in Figure 5.5, but instead is compared to the performance of the optimal partition assignments (represented by the second bar). The figure shows that the EMBARC algorithm achieves optimal speedup for all of the benchmarks listed. While these results allow comparisons of EMBARC to Panda's published results, these benchmarks are small and their use does not provide conclusive evidence of the effectiveness of the EMBARC algorithm on systems with both caches and SRAM.

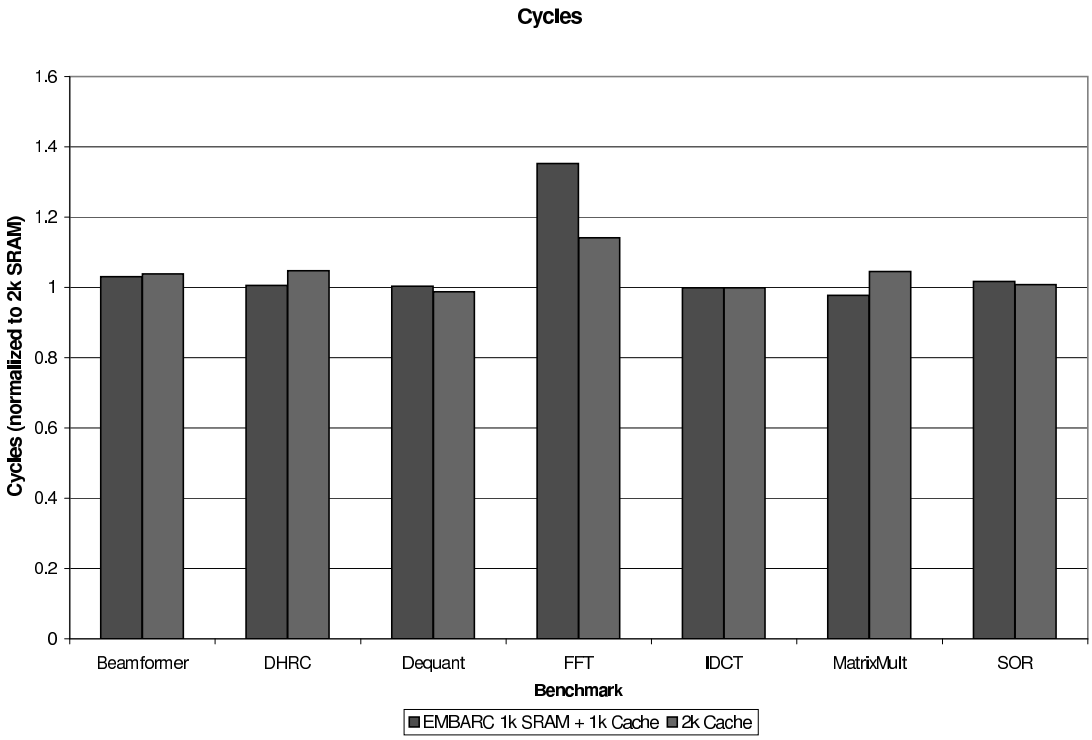

Figure 5.5: Cycle count for system with cache and SRAM totaling 2k

As a more thorough evaluation of the EMBARC algorithm, a more realistic set of benchmarks was selected. Figures 5.8 and 5.9 give the results of those experiments. The results

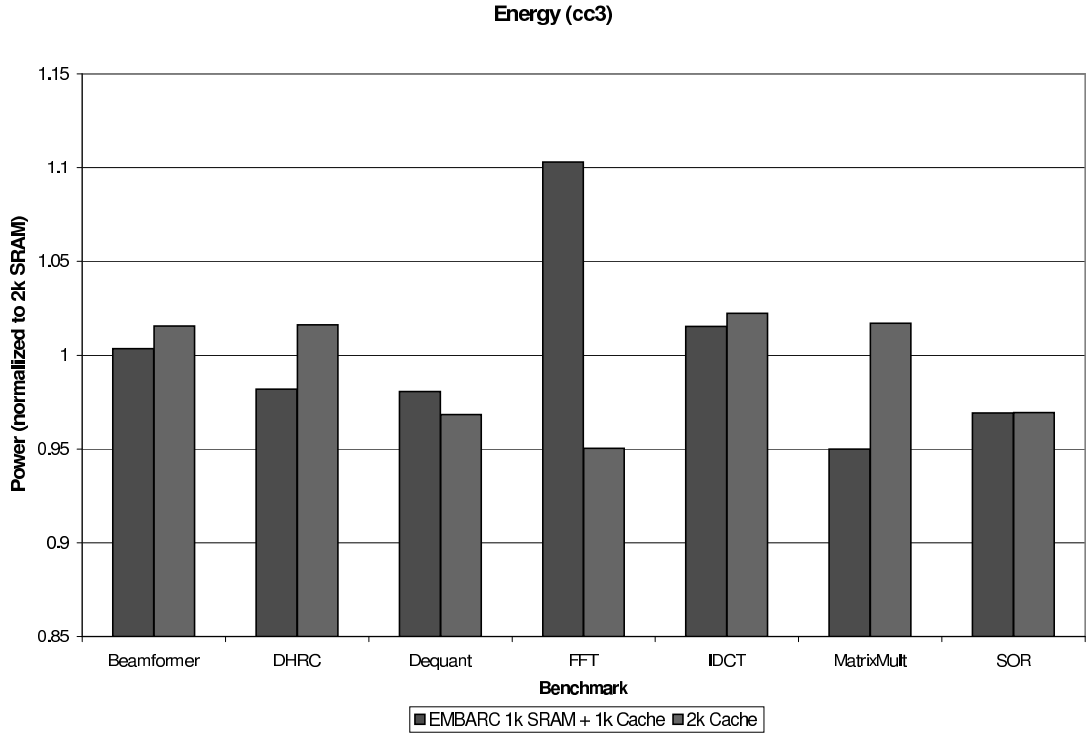

Figure 5.6: Energy for system with cache and SRAM totaling 2k

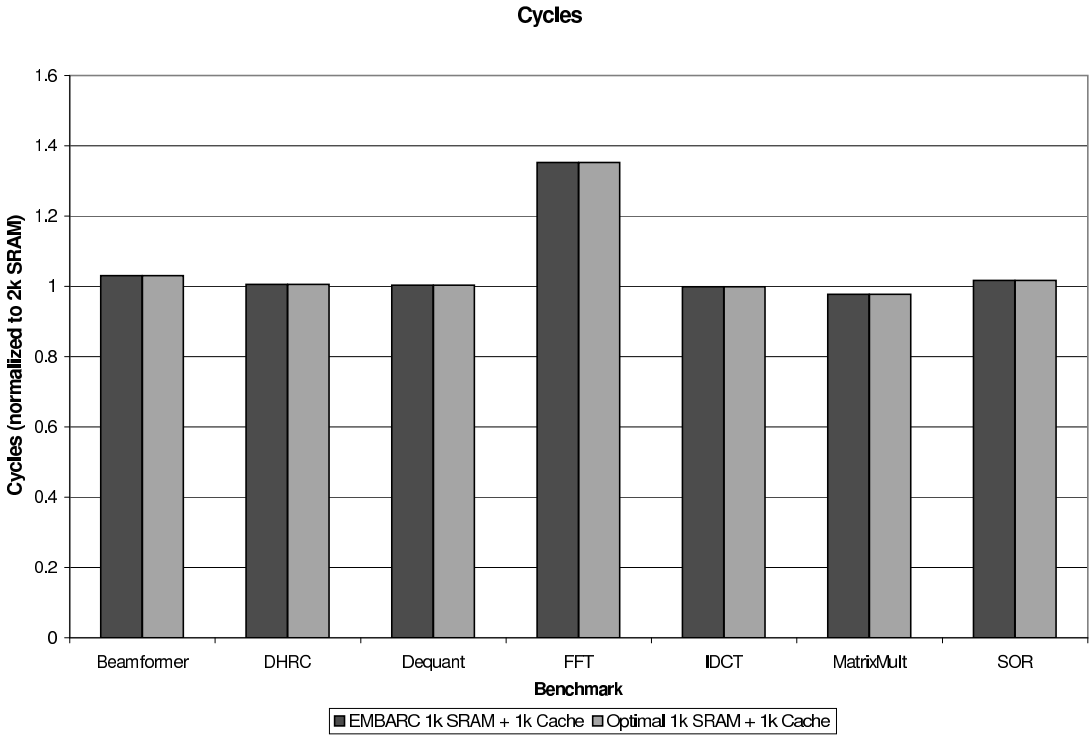

Figure 5.7: Optimal versus EMBARC for systems with 1k cache and 1k SRAM

demonstrate that the *2k cache* configuration and the *1k cache plus 1k SRAM* configurations are frequently very close to the same performance (*G721ML, trellis, adpcm, edge_detect, lpc, CRC32,* and *btoa*). For the remaining cases, the figures shows the 2k cache performing better on three benchmarks (*fft_kernel, dijkstra*, and *pegwit*) and worse on two benchmarks (*bmm* and *mpeg2.decode*). Note that the more realistic benchmarks show the same trend as the benchmarks used in Panda's work, the pure cache and mixed cache and SRAM configurations perform equally well. However, since the benchmarks are larger and more dynamic, it is more challenging to make effective use of the SRAM. The fact that the mixed SRAM and cache configuration still performs as well as the pure cache configuration indicates that the 1k of SRAM is being used at least as effectively as the extra 1k of cache. Thus, the figure shows evidence that EMBARC is generating effective partition assignments.

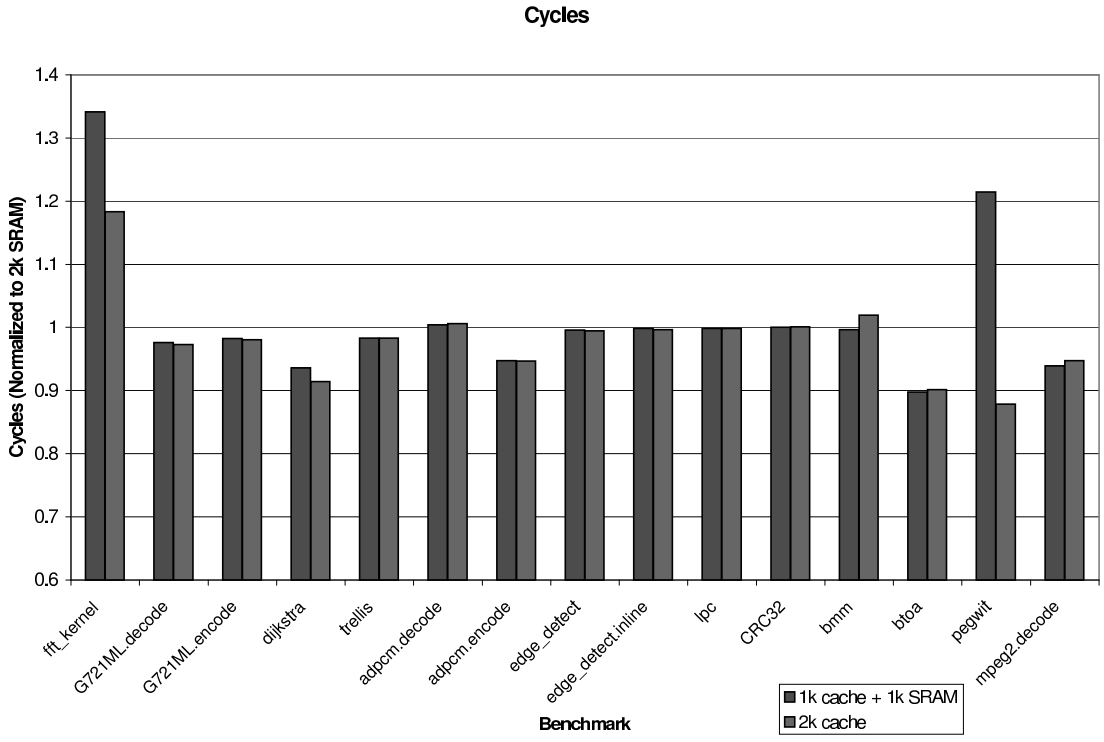

Figure 5.8: Cycle counts for system with cache and SRAM totaling 2k

To give further evidence that EMBARC is generating effective partition assignments, the results are compared to hand optimal solutions for a selection of benchmarks. Figure 5.10 gives these results. The first bar shows the performance of a system with 1k of SRAM

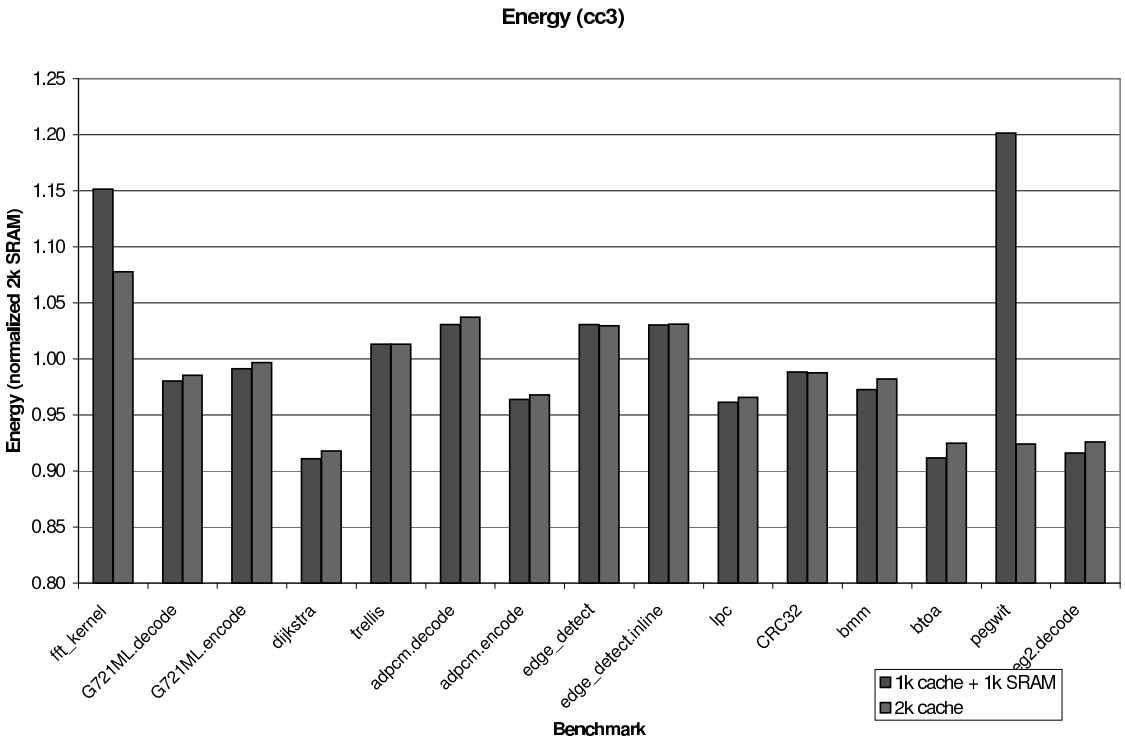

Figure 5.9: Energy for system with cache and SRAM totaling 2k

and 1k of cache when EMBARC is used to generate the partition assignments. The second bar shows the performance of the same system, but with hand generated optimal solutions. Both bars are again normalized to a system with 2k of SRAM. The figure shows the results are very close to optimal. In fact, EMBARC achieves the optimal solution for the *bmm* and *edge_detect* benchmarks. Based on this evidence, we assert that EMBARC generates effective partition assignments for systems that include SRAM and cache.

5.1.5 Two Caches

Even when a benchmark's variables cannot be partitioned to effectively take advantage of an on-chip SRAM, there may still be benefits to partitioning the on-chip resources. In particular, some benchmarks can take advantage of the additional bandwidth provided by partitioning. If the extra bandwidth was provided by partitioning the 2k cache into two 1k caches, the bandwidth could come without the disadvantages that an on-chip SRAM may have. Figures 5.11–5.12 show how using EMBARC to generate partition assignments for a

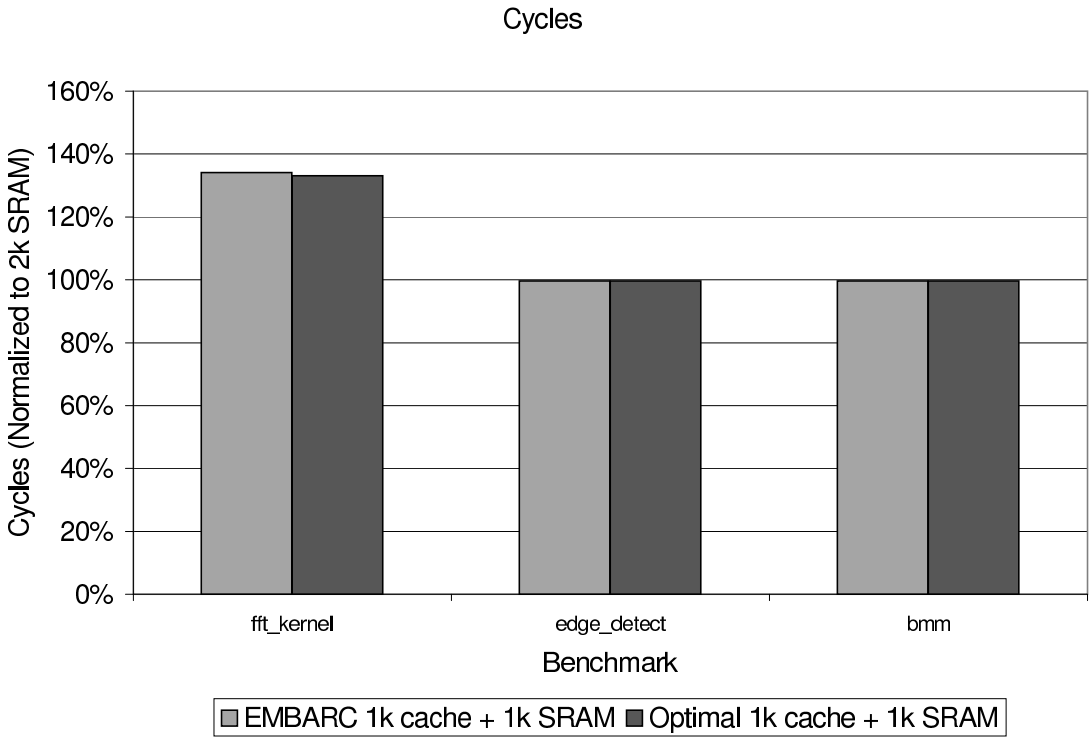

Figure 5.10: Optimal versus EMBARC for systems with 1k cache and 1k SRAM

system with two 1k caches would affect the energy and performance of the system on the benchmark set.

In Figure 5.11, the first bar represents the number of cycles used to execute the program and the second bar represents the amount of energy. Both bar are normalized to the values for a 2k cache. The figure shows that for most benchmarks, using two caches does not make sense. However, for some applications there are advantages. The *edge_detect* benchmark, in particular, has significant energy and runtime improvement. If the processor is being designed for a specific application, then being able to experiment with different memory hierarchies, even ones that may not make sense in general, would be a significant advantage. The primary advantage of the EMBARC framework over other work is that it can make partition decisions for a large range of memory hierarchies which allows a chip designer to evaluate different memory configurations.

Unfortunately, this data alone does not make it clear that EMBARC generates effective partition assignments. To determine the effectiveness of EMBARC, hand optimal solutions

were generated for three benchmarks: *fft_kernel, bmm,* and *edge_detect*. In Figure 5.12, the number of cycles and amount of energy for the optimal partition assignments are represented by bars two and four, respectively. EMBARC generated solutions that were within 3% of optimal for *bmm* and *edge_detect*. Unfortunately, *fft_kernel* does not show the same trend. After examining EMBARC and the solution it provides, it was found that the data profile shows little or no conflict between many of the variables in the program. Unfortunately, an outer loop in the application ensures that there is a large number of cache evictions that occur not because of conflicts, but because of limited cache capacity. Thus, EMBARC makes poor partition assignments because it lacks information about the enormous number of capacity misses that will occur in the caches. While it is believed that such situations are rare, future work should investigate ways to overcome this shortcomings.

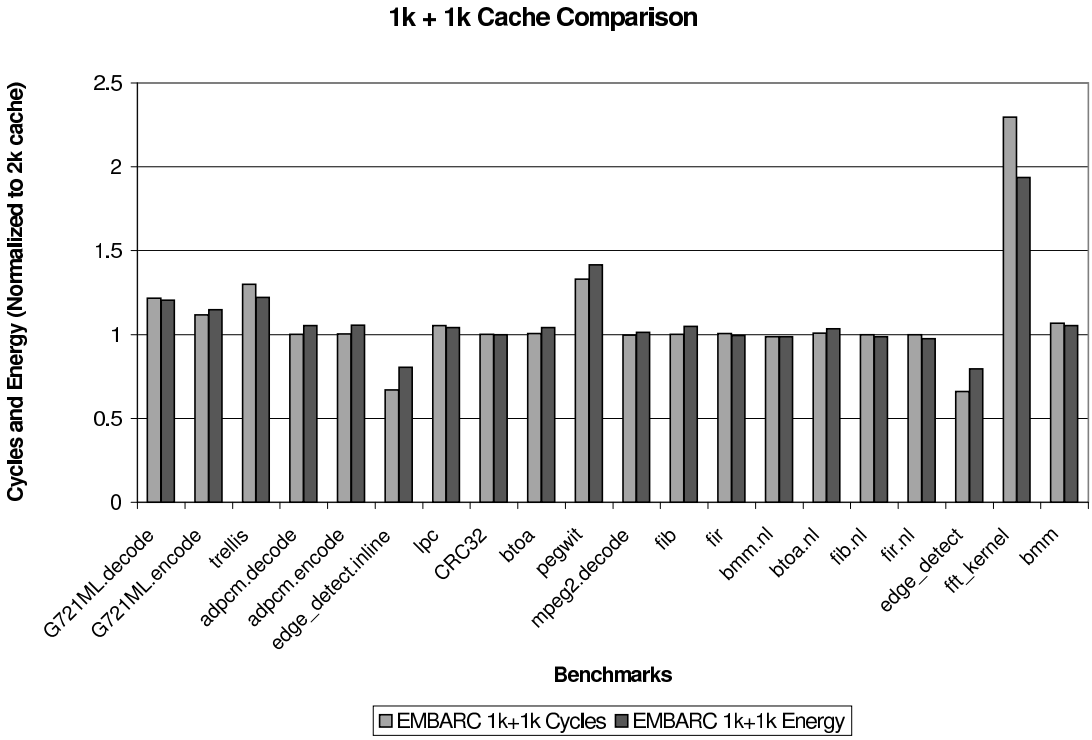

Figure 5.11: Runtime and Energy for systems with 2, 1k caches

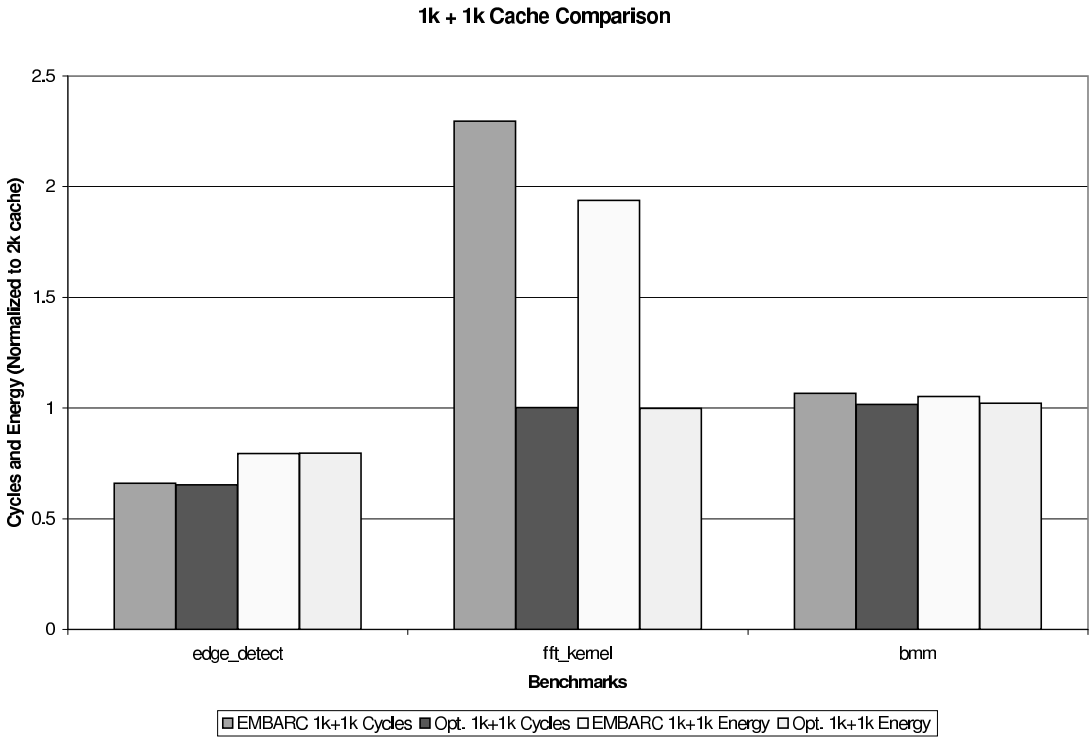

Figure 5.12: Runtime and Energy for systems with 2, 1k caches

5.1.6 Cache Bypassing

As a further demonstration that EMBARC generates effective partition assignments for a wide variety of memory hierarchies, a set of experiments were conducted for a machine that contains an on-chip SRAM, a data cache, and an optional cache bypass mechanism. The cache bypass mechanism is described in the partition description by specifying a partition that has no cache. By assigning a variable to the uncached partition, EMBARC indicates to the hardware that accesses to the variable are not to be cached. Thus, the cache bypass mechanism bypasses all references to the variable or no references to the variable. This bypass mechanism is in contrast to previous cache bypass work. Previous research instead allows individual memory operations to bypass the cache [55, 20]. Thus, a variable can be cached in one portion of the program, and uncached in other portions of the program. Consequently, comparing EMBARC's solutions to previous research would be difficult. Instead, EMBARC's solutions are compared to hand optimal solutions.

Figures 5.13–5.14 show the results of this experiment. In Figure 5.13, the first bar shows the performance of the solution generated by EMBARC for a memory hierarchy with 1k of SRAM and 1k of direct mapped cache and a bypass channel.. The second bar represents the optimal partition assignments for the configuration represented by the first bar. All bars are normalized to the performance of 1k of SRAM and 1k of direct mapped cache without the bypass channel. Now, first consider the *bmm* and *edge_detect* benchmarks. For these two benchmarks, the partition assignments provided by EMBARC are within 4% of optimal. Unfortunately, *fft_kernel* shows the same cache capacity problem as seen in the previous experiment.

For the remaining benchmarks in our suite, consider Figure 5.14. The first bar shows the performance of the solution generated by EMBARC for a memory hierarchy with 1k of SRAM and 1k of direct mapped cache with bypassing. The second bar shows the performance of a system with 1k of SRAM and 1k of fully associative cache and bypassing, but using the partition assignments from the experiment represented by the first bar. Again, all bars are normalized to the performance of 1k of SRAM and 1k of direct mapped cache. For many of these benchmarks (*dijkstra, trellis, adpcm.decode, adpcm.encode, lpc, CRC32, pegwit* and *fir*) there is little conflict left to avoid via cache bypassing, as evidenced by the fact that the fully associative cache gains no performance. In *mpeg2.decode*, we see that EMBARC has effectively used the bypass channel to achieve a 14% performance gain.

Thus, for eleven of fourteen benchmarks, EMBARC is generating effective solutions. As for *G721ML.encode* and *G721ML.decode*, the experiment is inconclusive and further work is needed to determine EMBARC's effectiveness in these benchmarks. However, there is strong evidence to support that EMBARC is generating effective partition assignments for a system including a cache bypass mechanism.

5.1.7 Sensitivity to Profiling Inputs

An issue with any algorithm that uses profiling information to guide its application is the sensitivity of the algorithm to the training inputs. Figure 5.15 gives the results of an

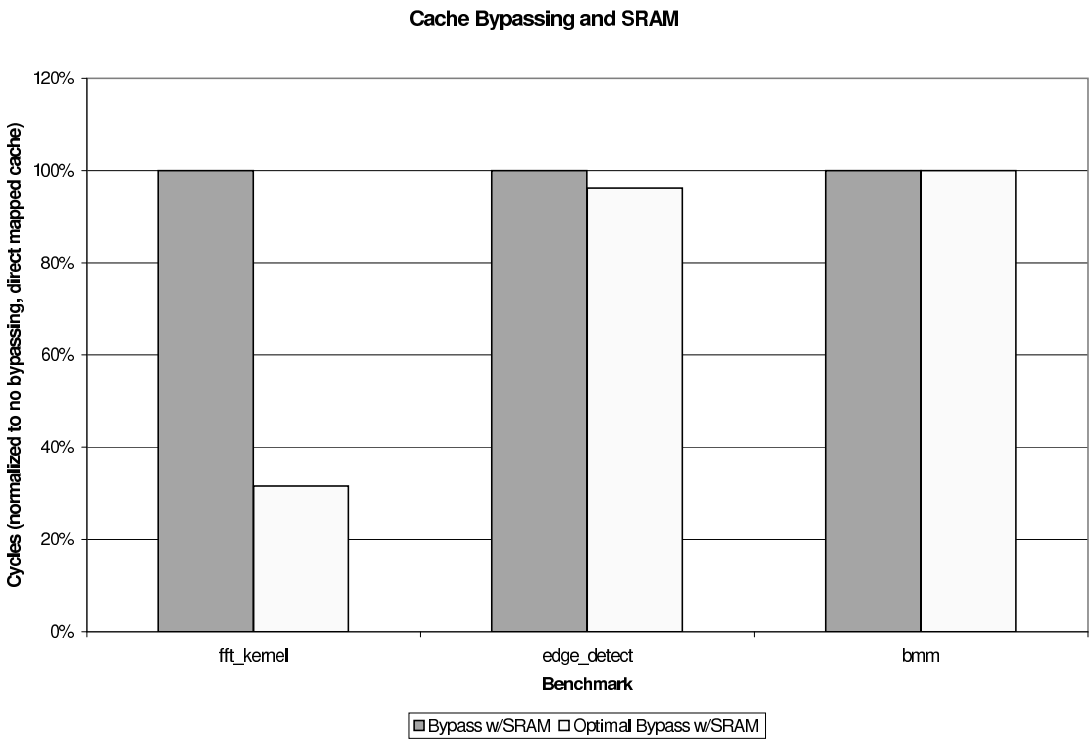

Figure 5.13: Optimal solutsions for systems with cache bypassing

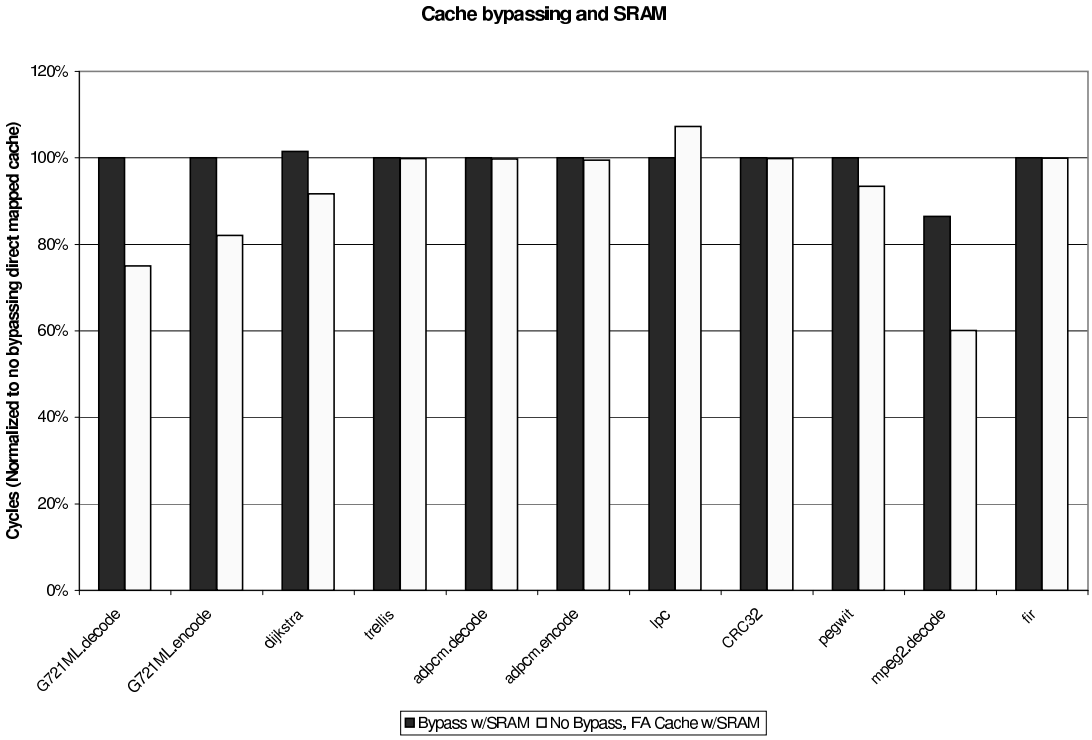

Figure 5.14: Runtime for systems with cache bypassing

experiment to determine the sensitivity of the EMBARC algorithm to the training inputs.

The graph contains the cycle count and energy consumption when an alternate profiling input (see Table 5.4) is used to partition three benchmarks—*CRC32*, *pegwit*, and *mpeg2.decode*. Each bar is normalized to the corresponding bar when the primary profiling input is used to partition the benchmark. (Note that all bars represent the program's behavior on the large input set, the differences result from different profiling input.) The graph has selected bars from the experiments in sections 5.1.2, 5.1.4, and 5.1.5.

Benchmark	**Input**
CRC32	*pegwit.lcex* – 96k VPO intermediate file
pegwit	*square.c* – 17k pegwit source file
mpeg2.decode	*chroma.m2v* – 37k m2v file

Table 5.4: Alternate profiling inputs

All the bars are near 1.0. This indicates that the alternate profiling input did not drastically affect the quality of the partition assignments. In fact, for *CRC32* the partition assignments are identical in each experiment. In *pegwit* and *mpeg2.decode*, the performance of the final application differs by less than 5%. Thus we conclude that the EMBARC algorithm is not highly sensitive to the profile input used.

5.2 Memory Hierarchy Estimation Quality

To evaluate how accurately MPRES quantifies the memory hierarchy performance, we compare its output with the actual results, obtained by simulation. Table 5.5 lists each of 80 different memory hierarchies used to evaluate our techniques. We chose to use memory hierarchies with 0-4KB of on-chip SRAM, 0-32KB on-chip cache, and 0-256KB of off-chip, L2 cache. Note that some configurations, such as configuration 50, may have 0KB of on-chip cache, but still access an off-chip cache. We chose 5 representative benchmarks from Table 5.1: *CRC32, dijkstra, adpcm.encode, pegwit*, and *mpeg2.decode*.

config num	SRAM	L1	L2	**config num**	SRAM	L1	L2
1	4	32	256	**41**	1	4	128
2	4	2	256	**42**	2	4	128
3	4	4	256	**43**	0	8	64
4	4	8	256	**44**	1	8	64
5	4	32	64	**45**	2	8	64
6	4	32	128	**46**	0	8	128
7	0	32	256	**47**	1	8	128
8	1	32	256	**48**	2	8	128
9	2	32	256	**49**	4	0	256
10	4	2	64	**50**	4	0	64
11	4	2	128	**51**	4	0	128
12	0	2	256	**52**	0	0	256
13	1	2	256	**53**	1	0	256
14	2	2	256	**54**	2	0	256
15	4	4	64	**55**	0	0	64
16	4	4	128	**56**	1	0	64
17	0	4	256	**57**	2	0	64
18	1	4	256	**58**	0	0	128
19	2	4	256	**59**	1	0	128
20	4	8	64	**60**	2	0	128
21	4	8	128	**61**	4	32	0
22	0	8	256	**62**	4	2	0
23	1	8	256	**63**	4	4	0
24	2	8	256	**64**	4	8	0
25	0	32	64	**65**	0	32	0
26	1	32	64	**66**	1	32	0
27	2	32	64	**67**	2	32	0
28	0	32	128	**68**	0	2	0
29	1	32	128	**69**	1	2	0
30	2	32	128	**70**	2	2	0
31	0	2	64	**71**	0	4	0
32	1	2	64	**72**	1	4	0
33	2	2	64	**73**	2	4	0
34	0	2	128	**74**	0	8	0
35	1	2	128	**75**	1	8	0
36	2	2	128	**76**	2	8	0
37	0	4	64	**77**	4	0	0
38	1	4	64	**78**	0	0	0
39	2	4	64	**79**	1	0	0
40	0	4	128	**80**	2	0	0

Table 5.5: Component sizes (in KB) for configurations

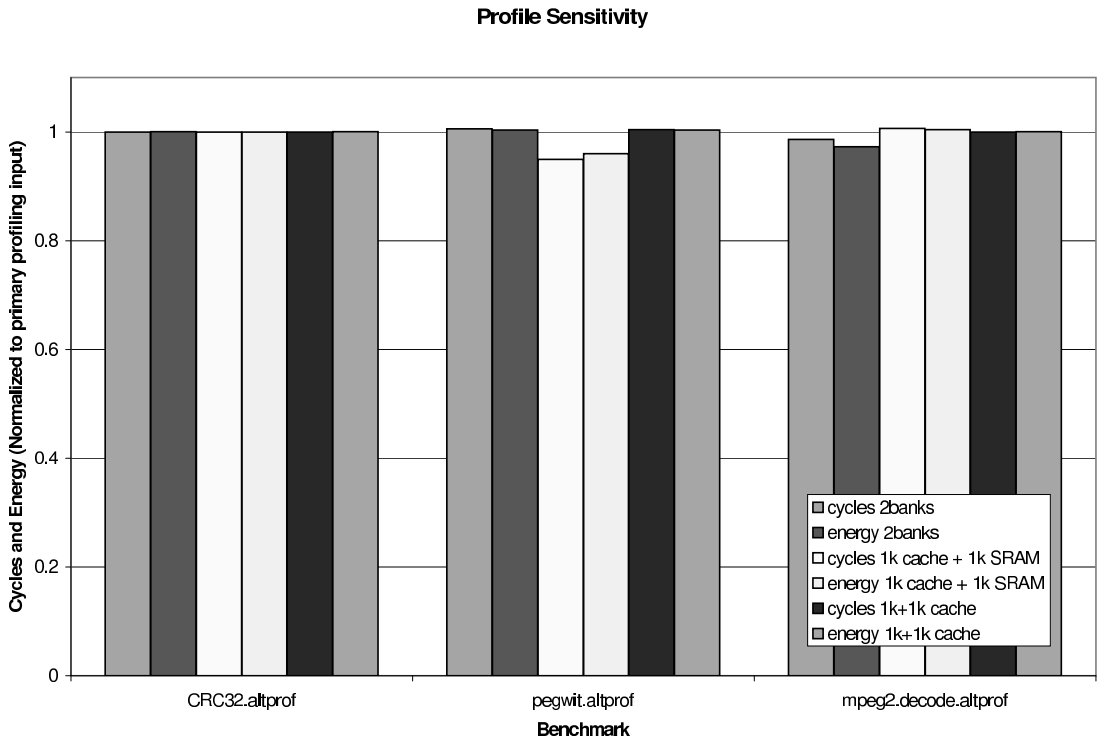

Figure 5.15: Profiling Sensitivity

5.2.1 Experimental Results

Figures 5.16–5.20 contain the results of evaluating MPRES for each of the 5 benchmarks. Each figure contains a line plotting estimated memory cycles for the benchmark and a line plotting the values obtained from a detailed simulation. Notice that the lines are plotted using different scales. The different scales are needed because MPRES has no way to estimate how long the program will run during the reference input. Since the reference runs execute longer than the training run, they cannot be plotted on the same scale. However, the key observation is that the trends are the same. The figures show that the estimate curve tracks very closely to the actual curve. However, we were able to produce the estimate curves in under 2 hours of compute time, while the curves corresponding to the actual memory time took over 200 compute hours to calculate.

To measure how closely the estimated memory cycles match the actual memory cycles, both the estimated values and the actual values were scaled so they are in the 0–1 range.

The scaling effectively makes each point a percentage of the maximum value. On this adjusted data set, the mean absolute error (MAE) and root mean square error (RMSE) were calculated. The MAE is sum of the absolute difference between the actual (adjusted) value and the estimated (adjusted) value. The RMSE is the square root of the sum of the square of the absolute differences.

Table 5.6 contains the MAE and RMSE numbers. The percent errors average only 1.5%. The small RMSE indicates that the errors are consistent, and few are off by more than the MAE. Figure 5.21 gives a histogram of the percentage errors. Each bar represents how many estimates deviate from the actual memory cycles by more than 1, 3, 5, 10, and 20 percentage points, respectively. The average bar shows over 70% of the data points deviate by less than 1 percentage point and 99% of the estimates deviate by less than 10 percentage points. The *CRC* benchmark does the best with all estimated memory cycles deviating by less than 3 percentage points.

Some points, however, have significant error. In particular, configuration 68 the *dijkstra* and *pegwit* benchmarks shows errors of 11.4 percentage points and 8.4 percentage points respectively. This increased error stems from configuration 68 itself and the dynamic nature of these benchmarks. Configuration 68 has no SRAM, no second-level cache, and a very small first-level cache. Since a program that has much dynamic memory allocation and large variables makes estimating conflict in the cache extremely difficult, the MPRES algorithm has difficulty determining the precise cache hit rate. Small errors in the estimated cache hit combined with a large discrepancy between a cache hit and miss (such as having no L2 cache) yield large errors in estimated run time. However, estimates within 10 percentage points for these cases are still useful.

The goal of MPRES is to provide estimates of memory hierarchy performance to system designers so they can select the most appropriate memory hierarchy, or they can select a small subset of memory hierarchies within a certain performance range. In the former case, the designer would want to compare two (or more) memory hierarchies using MPRES estimates and chose the memory hierarchy that best meets the application requirements. In

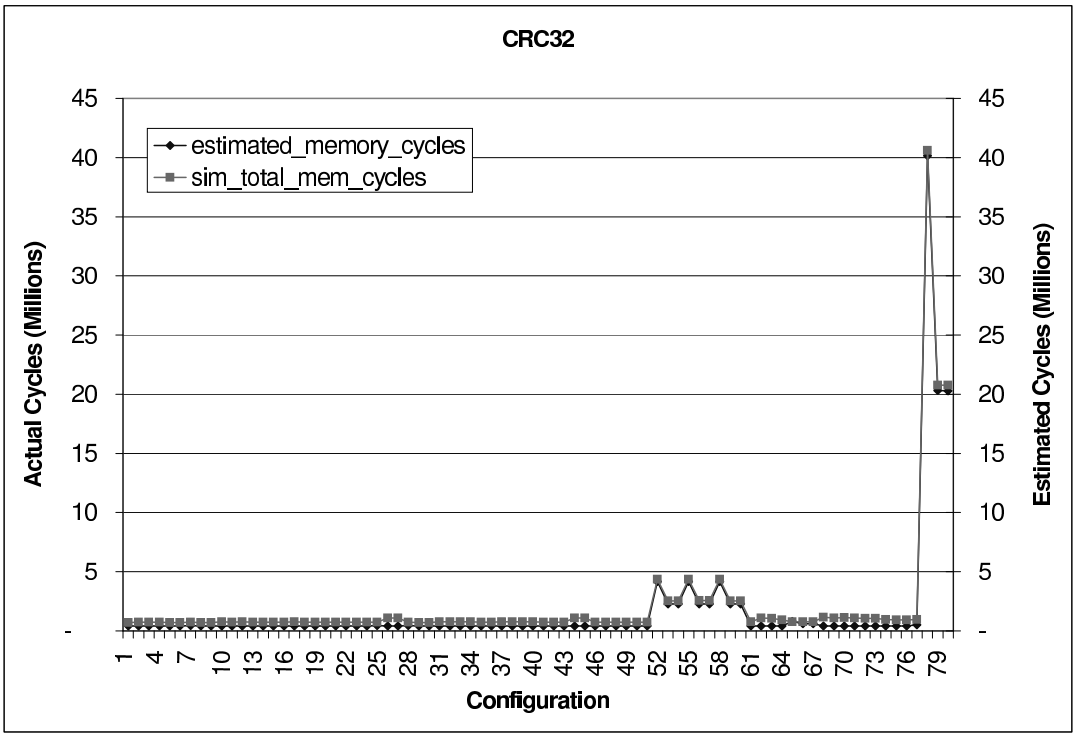

Figure 5.16: Estimated and Actual Memory Cycles for *CRC32*

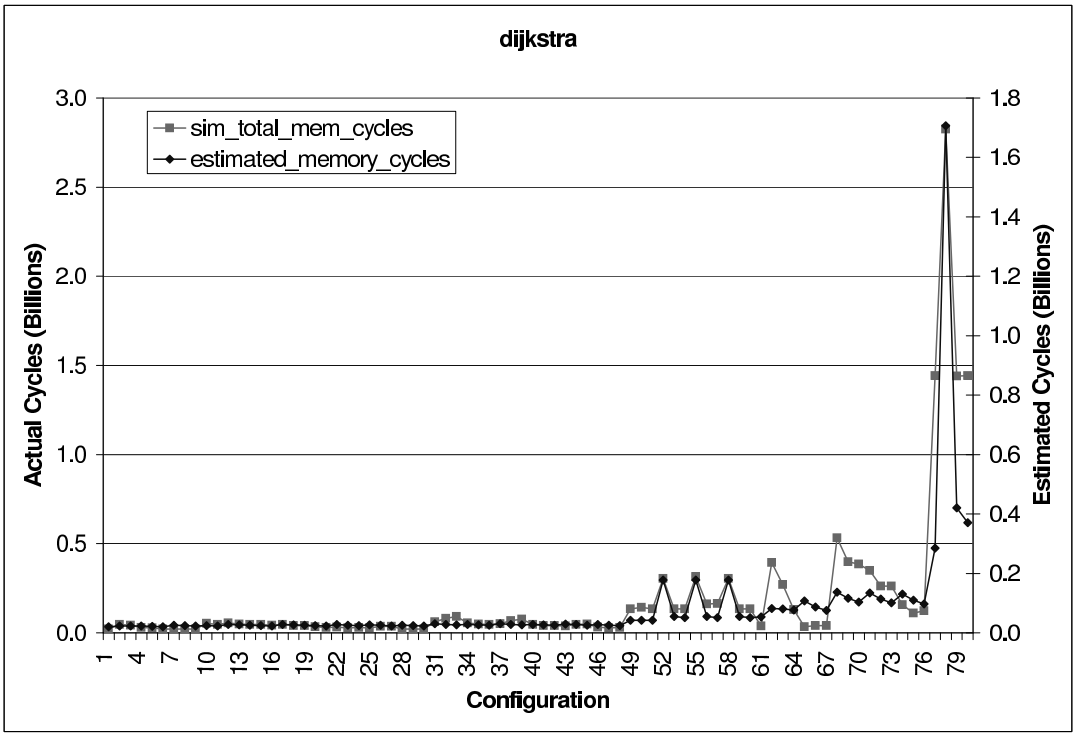

Figure 5.17: Estimated and Actual Memory Cycles for *dijkstra*

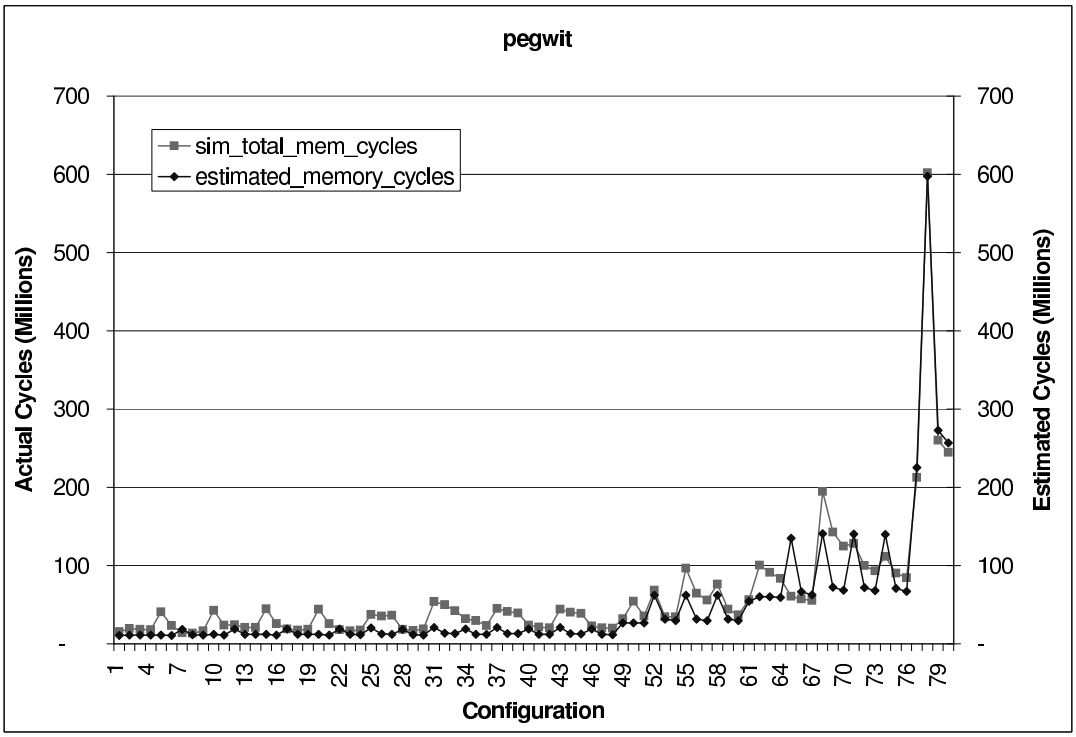

Figure 5.18: Estimated and Actual Memory Cycles for *pegwit*

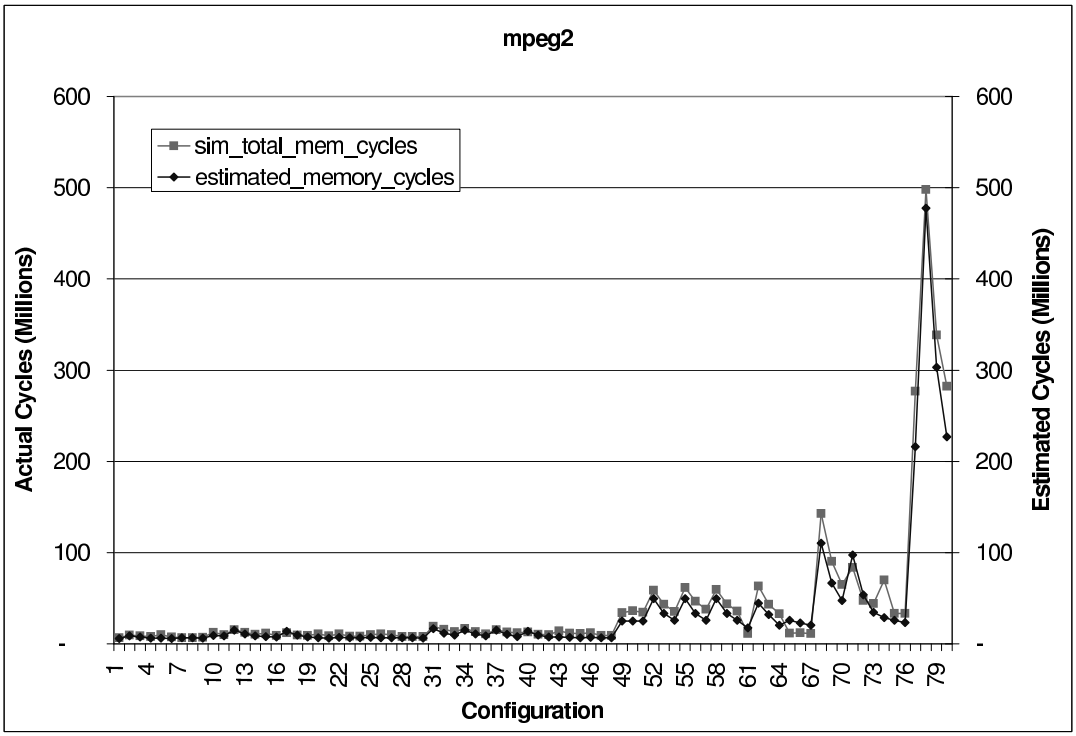

Figure 5.19: Estimated and Actual Memory Cycles for *mpeg2.decode*

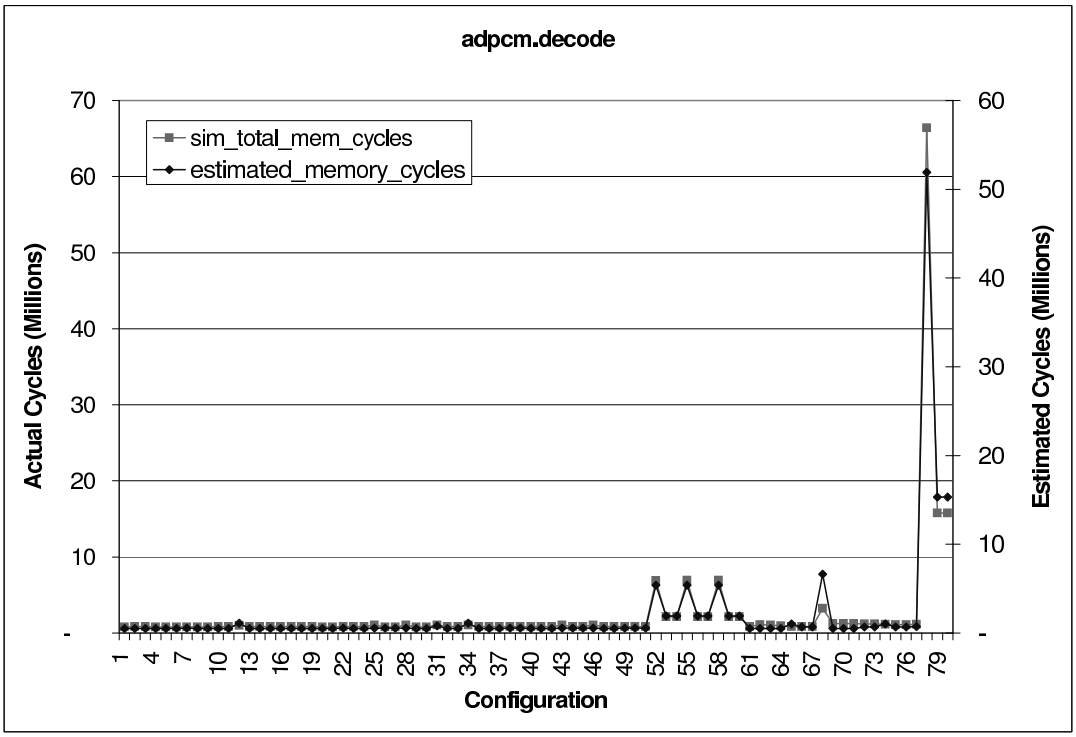

Figure 5.20: Estimated and Actual Memory Cycles for *adpcm.encode*

benchmark	MAE	RMSE
CRC32	0.27%	0.05%
dijkstra	2.66%	0.73%
pegwit	2.61%	0.41%
mpeg2.decode	1.39%	0.27%
adpcm.encode	0.45%	0.15%
average	1.5%	0.3%

Table 5.6: Mean Absolute Error and Root Mean Square Errors

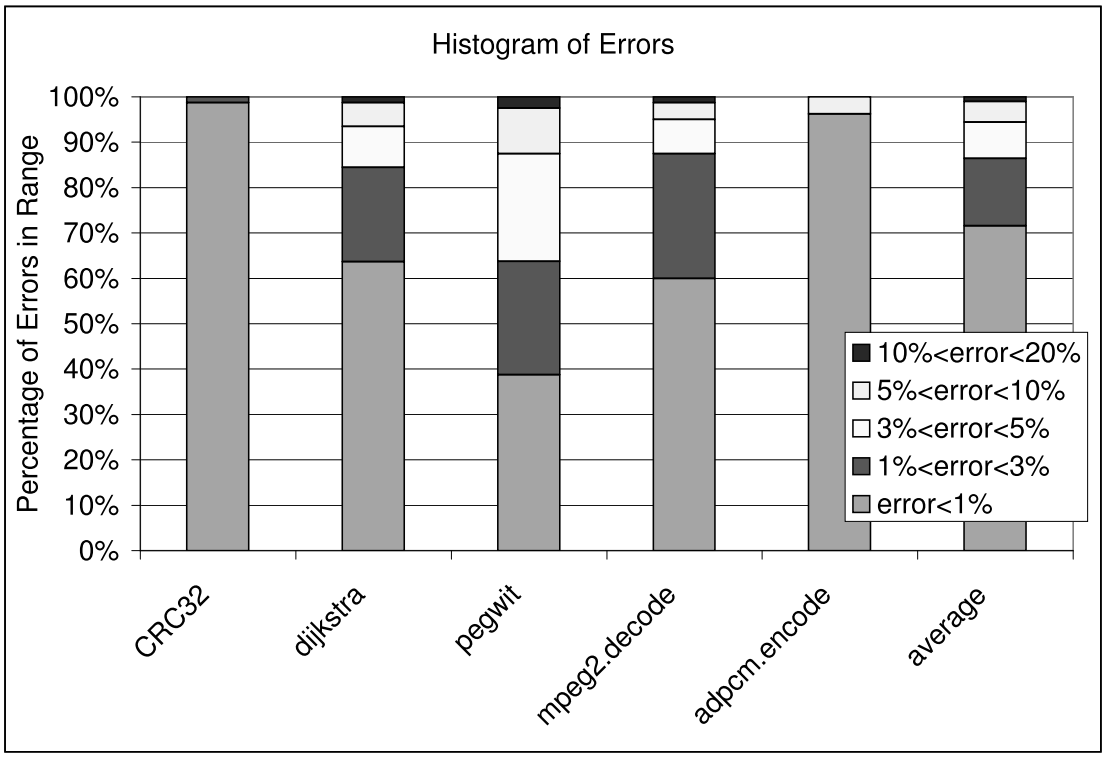

Figure 5.21: Histogram of Errors

the latter case, the designer would use a cycle-accurate (but slow) simulator to determine the which of the memory hierarchies is actually best satisfies the system specification.

While the previous graphs show that MPRES accurately estimates memory hierarchy performance, the graphs do not measure the *fidelity* of the estimates. The fidelity of the MPRES estimates is defined as the percentage of time the estimates of memory hierarchy performance can be used to select the memory hierarchy that actually provides the best performance in terms of cycle count.

Consider Figure 5.22a which shows a magnified view of the a section of a graph plotting both the MPRES estimates of cycles expended and the actual cycles expended determined by a cycle-accurate simulator. Comparing the estimates and choosing the one with the lowest cycle count results in the selection of the memory configuration that, in fact, does provide the best performance (in terms of cycles). Figure 5.22b shows a similar graph where choosing the estimate with the lowest cycle counts results in the selection of the memory

hierarchy that, in fact, does not provide the best performance (in terms of cycle count).

To measure the fidelity of the MPRES estimates, a pairwise comparison of all estimates was performed to determine the number of times the comparison of two estimates would result in the selection of a memory configuration that, in fact, was not the memory configuration that provided the best performance. Thus, for n configurations, there are $n(n-1)/2$ comparisons. An estimator has 100% fidelity if the estimator, when used to compare two memory hierarchies, always chooses the configuration that has the best actual performance.

The results of this pairwise comparison of the 80 estimates for the five benchmarks is presented in Table 5.7. The table shows the breakdown of correct, incorrect, and percent correct selections. The fidelity ranges from 73% to 90% across the benchmarks. Across all benchmarks the average is 80.7%.

The performance of two memory system configurations are frequently very similar, often only differing by a few percentage points. In these cases, it may not be important to discriminate between whether comparison of two estimates correctly selects the memory configuration that provides the best actual performance since there is only a small difference in performance.

To measure the fidelity of the estimates where it is important to correctly select the memory hierarchy that provides the best actual performance, pairs of memory hierarchies where the difference in actual cycle counts is less than the MAE were excluded. These percentages are reported in column 5. For the cases where fidelity matters, the range is from 90.8% to 97.1% with an average of 94.9%. Thus, it can be concluded that MPRES can be used with high confidence to select an appropriate memory configuration.

Benchmark	correct comparisons	incorrect comparisons	percent correct	restricted percent correct
CRC32	2,441	719	77.2%	95.1%
dijkstra	2,549	611	80.7%	90.8%
pegwit	2,611	549	82.6%	96.6%
mpeg2.decode	2,827	333	89.5%	97.1%
adpcm.encode	2,317	843	73.3%	94.8%
average	2,549	611	80.7%	94.9%

Table 5.7: Per benchmark memory hierarchy comparisons

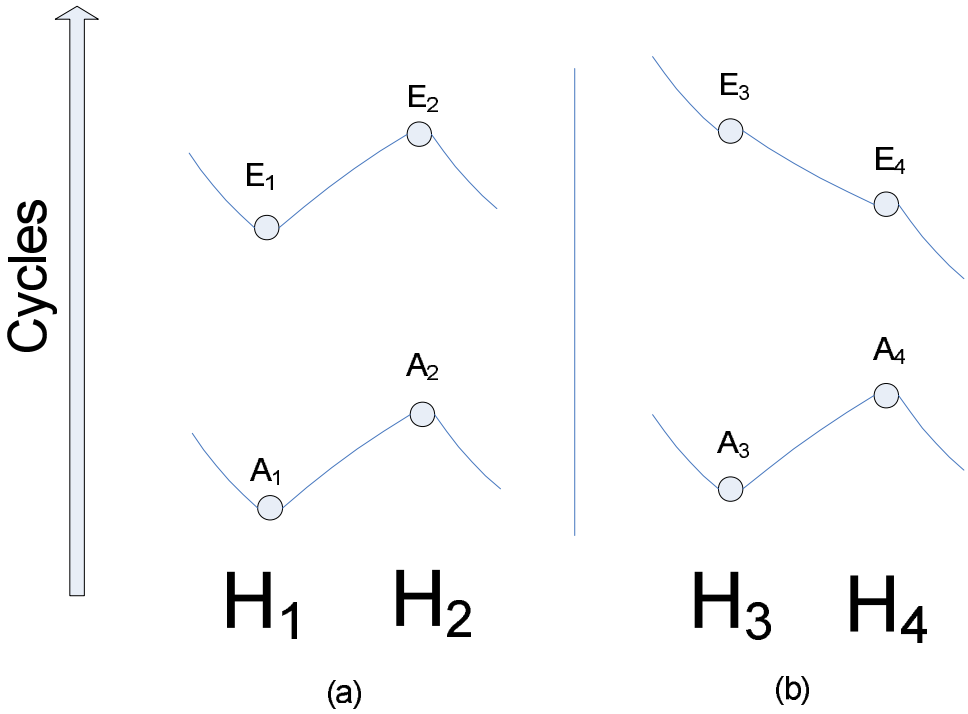

Figure 5.22: Example of correct and incorrect comparisons

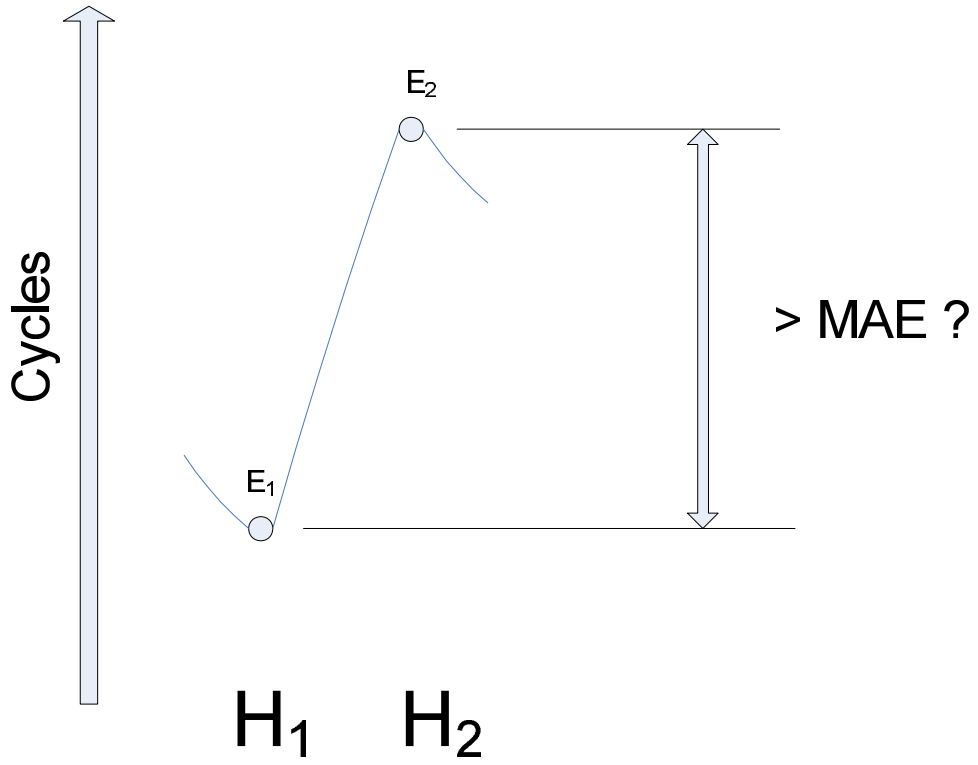

Figure 5.23: Example of restricting comparisons based on MAE.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Information technology advances have enabled the widespread use of *embedded* processors in the modern devices that make our lives easier. These devices, such as cellular phones and DVD players, are in very high demand and billions are sold each year. Consequently, it is important to make these devices as cost-effectively as possible. Saving just pennies on an embedded processor can make the difference between a profitable chip and one that loses money. Furthermore, embedded processors often have strict performance and energy consumption objectives. Failure to meet the objectives means that a competitor's processor may be used, thus achieving the objectives is paramount to the success of the processor.

One challenge is designing a memory hierarchy which provides data to the processor and meets strict performance and power objectives. A memory hierarchy that is fast enough may often be too expensive. Effectively designing a memory hierarchy is a problem in all computing systems. It is especially important because of the growing disparity between CPU and memory speeds. In fact, some researchers believe that all computer systems will soon be against the *memory wall*, or that the time to execute a program will be dominated by memory access time [57].

Because of the importance of selecting the most cost-effective memory hierarchy, embedded processor designers spend significant time determining the memory configuration

that best meets the needs of the target application(s) while still meeting the cost and energy goals. However, the embedded system designer cannot spend too long selecting the memory hierarchy. Designers must also meet strict time-to-market deadlines. If the processor takes too long to develop, a competitor's chip may be used.

Consequently, the embedded processor designer has the challenging job of designing a cheap, highly effective memory hierarchy in a short time frame. One type of memory hierarchy that can be used to help meet these goals is a *partitioned* memory hierarchy. In a partitioned memory hierarchy, the memory in the system is divided into separate memory banks. The processor can potentially access each memory in parallel.

Partitioned memory hierarchies have many benefits. By accessing partitions in parallel, the effective access time can be reduced. Having more than one partition also increases the bandwidth to the processor. By accessing smaller partitions, energy can be saved. However, partitioning can have drawbacks too. If the target applications are unable to effectively use all the partitions, some resources are wasted. Furthermore, to access more than one partition, the processor must have extra logic. The cost of this extra hardware must be amortized by the benefits of the extra partitioning. Lastly, by allowing more partitions, the size of the search space grows exponentially.

To help the embedded system designer ensure that these drawbacks do not outweigh the benefits of partitioning, algorithms are needed to help the designer select the best hierarchy for their needs. One algorithm that is needed is one that supports quickly recompiling the program to take advantage of an arbitrary memory hierarchy. An important component of compilation for a partitioned memory hierarchy is assigning variables to memory partitions. This dissertation describes an efficient, memory bank assignment algorithm designed for retargetable compilers, named EMBARC.

EMBARC's inputs are a memory partition description, the target program, and a data profile. The data profile is generated by running the program with a sample input on a simulator which collects "live ranges" for each variable in the program. Each live range is terminated if the variable is not accessed after a specified number of cycles. In this work,

a live range is terminated after 1,000 cycles.

To assign variables to memory partitions, EMBARC first estimates the access time of each partition. It does this by assuming cache hit rates will be 90%, then estimates the average access time for each cache in the memory hierarchy. Finally, an estimated access time for each partition is calculated. EMBARC then considers stack, heap, and global variables from most frequently used to least frequently used. For each variable, EMBARC first calculates the cost of assigning the variable to each partition. The cost of placing the variable in a partition is based on how frequently the variable will conflict with other variables already assigned to that partition and how long the partition will take to satisfy the requests for that variable. The net result is that variables that conflict frequently will be pushed to different partitions, and less frequently conflict variables will reside in the same partition. Also, the most frequently referenced variables get first priority for the fastest partitions.

EMBARC's effectiveness is evaluated by comparing its performance to algorithms designed to perform well on specific memory hierarchies. On systems with multiple banks of DRAM, EMBARC achieves a 41.8% speedup over an ideal machine. Previous work shows a 42.4% speedup in similar experiments. We conclude that EMBARC generates solutions that are 99% as effective as an algorithm specifically designed to handle multiple banks of DRAM. On systems with a mix of SRAM and DRAM partitions, EMBARC's solutions are very close to solutions provided by an optimal ILP solver. In cases where hand-optimal solutions were feasible, EMBARC generated the optimal partition assignment. Lastly, on machines with SRAM and cache, EMBARC and algorithms specifically designed for SRAM and caches both generate solutions in which mixed SRAM and cache perform similar to pure cache memory hierarchies.

Once variables are assigned to partitions, the embedded systems designer needs to be able to evaluate how effectively the candidate memory hierarchy performs for the program. Ideally, a processor would be built in silicon and the program would be executed. However, there are many potential candidates for the memory hierarchy, perhaps hundreds

or thousands. Consequently, building a processor for each is too expensive and too time consuming. Instead, designers must rely on an estimation of the actual hardware. One common approach is to use a simulator to gain performance metrics. However, simulators are slow—often two to one hundred times slower than native hardware. If five applications need to be evaluated on 100 memory hierarchies and each simulation takes 2 hours, over 1,000 hours of compute time would be required. With 10 CPUs executing simulations, over four days would be needed for a small test suite to be fully evaluated. Worse yet, the target application(s) can evolve late in the design cycle. Design goals may not be met if re-evaluating the test suite requires 4 days. The embedded system designer needs a faster technique for evaluating the performance of the memory hierarchy.

To eliminate embedded system designer's need to rely on time-consuming simulations, this dissertation also developed a technique, named MPRES, for estimating the performance of a memory hierarchy given a program and a data profile. The data profile is the same as the one used in the partition assignment phase, described above. MPRES starts by calculating an effective cache size for each cache, in a top-down manner. The effective cache size is based on the actual cache size and the amount of conflict in the cache. More conflict reduces the effective cache size, while less conflict increases it. With the effective cache size calculated, MPRES next estimates a hit rate for each cache. With estimated hit rates for each cache, calculating the average access time for each cache and each partition is straightforward. Finally, MPRES emits a total memory access time based on the access to each partition, and the estimated partition access time.

To evaluate the MPRES algorithm, the estimations generated were compared with the results of simulations. Five representative benchmarks were chosen from common embedded benchmark suites, *CRC32, dijkstra, pegwit, mpeg2.decode*, and *adpcm.encode*. Eighty unique memory hierarchies were tested with varying amounts of first-level cache, second-level cache, and SRAM. The simulations represent over 2 weeks of compute time, while the estimations took less than 2 hours of processor time. The total memory access time from the estimation phase was compared to the total memory access time determined by simulation. The

results show that the estimations track very closely to the results of the simulations. After normalizing, the mean absolute error (MAE) is within 1 percentage point and the root mean square error (RMSE) is likewise very small. We conclude that MPRES generates estimates that are accurate, and beneficial to an embedded system designer.

The solutions in this dissertation provide a comprehensive solution to the memory hierarchy design problems faced by embedded system designers. Together, MPRES and EMBARC allow an embedded systems designer to quickly and effectively evaluate a broad range of partitioned memory hierarchies, thus allowing the embedded system designer to more fully exploit partitioned memory hierarchies while meeting their design objectives.

6.2 Future Work

Although the work presented here significantly advances the state of the art, there are still areas which need to be researched. These areas include combining partition assignment with data layout (Section 6.2.1), extensions to EMBARC for better partition assignments with regard to cache capacity (Section 6.2.2), choosing dynamic partition assignments (Section 6.2.3), researching the instruction paradigm (Section 6.2.4), evaluating the usefulness of partitioning in desktop and server machines (Section 6.2.5.), extensions to the partition description language (Section 6.2.6), faster access to data via duplication (Section 6.2.7), supporting multiple execution stacks in a system (Section 6.2.8), and dealing with context switches (Section 6.2.9).

6.2.1 Partitioning and Data Layout

The benefits gained by data layout optimizations are mostly from reduced conflicts in caches and maximizing the hardware's exploitation of locality. Thus, the effectiveness of data layout optimizations depends on the application's access patterns. Separating highly conflicting variables into separate partitions is beneficial for all applications, but may be too aggressive. Depending on the application's access patterns, it may be possible to

carefully map highly interfering data into the same partition so that conflicts are minimized. Using a data layout technique to reduce conflicts could dramatically effect the shape of a good partition assignment. It would be interesting to consider data layout and partition assignment as one combined phase.

One approach to combining data layout and partition assignment would be to assign variables to each partition much like EMBARC does. However, the cost metric would vary based on how well a data layout algorithm can place the variable in a partition. For example, if variables A, B, and C were already assigned to a partition, P, then assigning variable D to partition P would require running a data placement algorithm to estimate the performance of partition P with variable D assigned to P. Once D is considered for all possible partitions, it is assigned to the partition where data layout does the best job of minimizing conflicts and maximizing locality.

6.2.2 Cache Capacity Extensions

Sections 5.1.5 and 5.1.6 discussed a situation with the *fft_kernel* benchmark that caused EMBARC to generate ineffecient partition assignments. The ineffecient assignments were caused by EMBARC's failure to account for capacity misses in the cache. We believe that extensions to the locality profiling technique (to include the number of unique bytes addressed by each psuedo-live range) and the EMBARC algorithm to include information (to make effective use of the extended dynamic profile) can help to eliminate the anamoly caused in this case. Further work is needed to know how common such cases are, and whether the extra work is necessary.

6.2.3 Dynamic Partition Assignment

EMBARC and MPRES currently assume that partition assignments do not change during the execution of the program. It is widely believed that programs go through different phases [46, 45]. Unfortunately, a single static partition assignment may be optimal for the entire program while being sub-optimal for each phase. If partition assignments could

be made for each phase of the program, and dynamically applied, the benefits could well outweigh the cost of applying the decisions dynamically.

Previous work has identified that certain program points are obvious choices for changing partition assignments [52]. Loop preheaders and function calls are commonly chosen. If the program regions are sorted topologically by execution order, EMBARC could make assignments in a top-down fashion. The cost of placing a variable would need to be adjusted to reflect the cost of copying the variable when its partition assignment changes. The cost would need to be adjusted based on how frequently the transition happens, and the runtime cost of copying the variable (small variables take less time to copy than larger variables, for example).

6.2.4 Partitioning with the ISA Paradigm

When using the ISA paradigm, the compiler needs to identify the variables that each memory instruction in the program can access. Unfortunately, some memory instructions will likely access more than one variable. These instructions present a problem when making partition assignments. If a memory instruction accesses both variable v_1 and v_2, then the variables either need to be assigned to the same partition, or use an instruction that can access more than one partition. If the machine has memory instructions that can access all partitions, then no restriction is placed on the assignment. However, this is more like the address paradigm, and most machines either would not have such an instruction, or there would be a significant cost (delay and energy) associated with using it in an ISA paradigm instruction set. Consequently, use of such an instruction should be strongly avoided. It would likely be better to assign variables v_1 and v_2 to the same partition. By coalescing v_1 and v_2 before the partition assignments are made, the assignment phase can assign a partition to the variables simultaneously.

Some memory instructions, however, are particularly hard to analyze and the compiler may not be able to determine which variables are accessed. Often, these memory references are seen as addressing almost every program variable. If the compiler coalesces most

variables during the coalescing phase, partition assignment is significantly restrained and prohibited from making an effective assignment. Difficult to analyze memory instructions may be candidates for the more expensive type of memory operation that can access multiple partitions. A thorough evaluation of coalescing and techniques to determine which memory operations should trigger coalescing is needed before the effectiveness of these techniques is known.

6.2.5 Partitioning for Desktop and Server Applications.

This work has focused on how to effectively make partition assignments for embedded applications. Many of the advantages of partitioning could be exploited by desktop and server processors as well. Just as in embedded systems, the compiler must produce code that effectively uses the partitions in the processor. Further research needs to determine what types of memory partitioning can be effectively used, what techniques are appropriate for profiling very large applications, and whether static or dynamic partition assignments are more effective.

6.2.6 Partition Description Language Extensions

The partition description language is ideal as an input for EMBARC. Its simple structure enables easy readability, writeability, parseability for the language. Unfortunately, the level of detail may be insufficient for other uses. Some features are missing that would be necessary to perform a cycle-accurate simulation for the most common memory hierarchies. Store buffers, cache sub-blocking policies, TLBs, and interblock DRAM access time characteristics are lacking from the partition description language. Adding such features to the language may make it possible to use the language for other tasks, such as accurately simulating an arbitrary memory hierarchy in SimpleScalar.

As far as we know, no other work has attempted to take such features into account when assigning variables to memory partitions. It may be possible to make better memory partition assignments if such features are considered. Additional research needs to deter-

mine whether using information about more complex memory hierarchy features improves the resulting partition assignment.

6.2.7 Data Duplication

As was evident in evaluation of the EMBARC algorithm (See Section 5.1.2), some applications need data duplication to achieve the best speedups. Consider assigning variables to 2, equal, single-ported DRAM partitions for the source code in Figure 6.1. If variable a is

```
for(i=0;i<N;i++)
{
    c[i]=a[i]+a[2*i];
}
```

Figure 6.1: Sample source code demonstrating the importance of data duplication

assigned to either partition exclusively for the duration of the loop, the two references must be satisfied sequentially. If the DRAM banks have extra room, then duplicating variable a can result in satisfying the memory references in parallel, almost halving the time for the loop.

However, deciding when and how to duplicate variables is a difficult problem and there are many costs associated with data duplication. Alias analysis is necessary to determine which loads to a duplicated should access which partition, there must be sufficient DRAM for the duplication, and all copies of the variable need to be updated on a write. The problem is further complicated if the duplication is only to be in effect for a portion of the program. Thus, considering the benefits and difficulties associated with duplicating variables, there is much that needs to be researched in this area.

6.2.8 Multiple Execution Stacks

The research presented in this dissertation assumes that a single register is set aside for a stack pointer, and points to default partition. The activation stack is held entirely in that partition. When the compiler can statically determine that a variable cannot be involved

in direct recursion, the variable may be promoted to a global variable so that it may reside in another partition.

However, previous work has demonstrated that having more than one stack can be beneficial [7]. Unfortunately, for a general memory hierarchy multiple execution stacks cause difficulty. The cost of having a separate stack includes having a separate stack pointer, which could be used for a scalar value if it were not allocated as a stack pointer. If the stack is to be partitioned automatically, the compiler needs to know which partitions are suitable candidates, when the costs of allocating a stack in a partition is beneficial, and what to do if the stack in the partition overflows. Clearly, there are benefits to partitioning the stack, but there are many significant obstacles to overcome. More research is needed to determine when and how to partition the stack.

6.2.9 Context Switches

The work in this dissertation and most of the previous work assumes that the memory hierarchy is dedicated to a single application. By taking the contexts of multiple programs into account, any single-program solution can generate a partitioning for multiple programs. However, if the list of processes is arbitrary or unknown *a priori*, such solutions are not feasible. The OS can effectively manage DRAM by swapping pages to and from disk, but on-chip SRAM is more difficult to deal with. To run a program that is compiled to use the SRAM effectively, the program must have access to the SRAM. However, when a context switch occurs and another program needs to use the SRAM, swapping the contents of the SRAM to DRAM or disk is prohibitively expensive. Finding effective ways to deal with SRAM in a multiprocess system is a difficult problem that needs further research. If general solutions are found, SRAM may even be a viable solution for desktop and server machines.

6.3 Contributions

In summary, this dissertation has significantly advanced the state of the art in memory partitioning software. Algorithms to help embedded system designers and compiler writers effectively deal with partitioned memory hierarchies have been presented. In particular, this dissertation sets forth the following major contributions:

- A technique was presented for collecting data profiles from a program in which each variable has a list of pseudo-live ranges.

- A memory hierarchy description language was presented. The language is suitable for describing the memory hierarchy of a machine in enough detail that partition assignment and runtime estimation are possible, but simple enough for embedded system designers and compiler writers to use without undue burden.

- Also presented was a technique for assigning program variables to memory partitions given a memory hierarchy description and the program. This technique is called EMBARC. Previous techniques have assumed a fixed memory hierarchy. This assumption severly limits the use of previous algorithms.

- Next, the dissertation presented a technique to estimate how effectively a program will execute for a given memory hierarchy. This technique is called MPRES. MPRES can be used to quickly evaluate a large set of memory hierarchies for a set of target application. Previously, embedded system designers needed to rely on costly simulation to gather such information.

- Lastly, EMBARC and MPRES are evaluated using a wide benchmark suite and a wide variety of memory hierarchies. EMBARC was found to compare favorably to algorithms designed specifically for a particular memory hierarchy. MPRES was found to give results useful to an embedded system designer. This evaluation also demonstrates the need for suitable benchmarks when evaluating memory heirarchy algorithms.

Bibliography

[1] Tom's hardware website. World Wide Web, `http://www.tomshardware.com/motherboard/20040119/index-01.html`.

[2] Trimaran test suite. World Wide Web, `http://www.trimaran.org/`.

[3] University of Toronto DSP benchmark suite. World Wide Web, `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.tar.gz`.

[4] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 23th Annual International Symposium on Microarchitecture*, pages 139–152, 1993.

[5] S.G. Abraham and S.A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Proceedings of the 32th Annual International Symposium on Microarchitecture*, pages 114–125, 1999.

[6] D. W. Anderson, F. J. Speracio, and R. M. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[7] O. Avissar and R. Barua. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[8] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 34–43. ACM Press, 2001.

[9] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computer Survey*, 26(4):345–420, 1994.

[10] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78. ACM Press, 2002.

[11] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 329–338, 1988.

[12] D. G. Bradlee, S. J. Eggers, and R. R. Henry. The effect of RISC performance of register set size and structure versus code generation strategy. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 330–339, 1991.

[13] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94. ACM Press, 2000.

[14] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, 1996.

[15] B. Calder, C. Krintz, S. John, and T. M. Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149. ACM Press, 1998.

[16] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, 1992.

[17] A. Capitanio, N. Dutt, and A. Nicolau. Partitioning of variables for multiple-register-file VLIW architectures. In Dharma P. Agrawal, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 1: Architecture*, pages 298–301, 1994.

[18] A. Capitanio, N. Dutt, and A. Nicolau. Toward register allocation for multiple register file VLIW architectures. Technical Report ICS-TR-94-06, University of California, Irvine, Department of Information and Computer Science, 1994.

[19] C. Chi and H. Dietz. Improving cache performance by selective cache bypass. In *Proceedings of the 22nd Hawaii International Conference on Systems*, pages 277–285, 1989.

[20] C. Chi and H. Dietz. Unified management of registers and cache using liveness and cache bypass. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 344–353. ACM Press, 1989.

[21] T. Chiueh. Sunder: A programmable hardware prefetch architecture for numerical loops. In *Proceedings of the SIGPLAN 1994 Conference on Supercomputing*, pages 488–496, 1994.

[22] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 138–147. ACM Press, 2000.

[23] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[24] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite, 2001.

[25] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[27] T. Horel and G. Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. In *IEEE Micro*, pages 73–85, 1999.

[28] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, October 1996.

[29] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 303–312. IEEE Computer Society TC-MICRO and ACM SIGMICRO, 1995.

[30] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the SIGPLAN 1990 International Symposium on Compiler Construction*, pages 364–373, 1990.

[31] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Conference on Design Automation*, pages 690–695. ACM Press, 2001.

[32] R. E. Kessler. The Alpha 21264 microprocessor. In *IEEE Micro*, pages 24–36, 1999.

[33] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–51, 1991.

[34] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[35] S. Lee, J. Lee, S. L. Min, J. D. Hiser, and J. W. Davidson. Code optimizations for a dual instruction set processor based on selective code transformations. In *Proceedings of 7th International Workshop on Software and Compilers for Embedded Systems*, pages 33–48, 2003.

[36] H. Lin and W. Wolf. Co-design of interleaved memory systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*, pages 46–50. ACM Press, 2000.

[37] S. McFarling. Cache replacement with dynamic exclusion. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 191–201, 1992.

[38] D. R. Miller and D. J. Quammen. Exploiting large register sets. *Microprocessors and Microsystems*, 14(6):333–340, 1990.

[39] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 103–114, 1998.

[40] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaeart, 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):149–206, April 2001.

[41] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, 2000.

[42] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *The 18th Annual Workshop on Microprogramming*, pages 109–116, 1985.

[43] J. Robertson. Intel hints of next-generation security technology for mpus. *EE Times*, Sept. 10 2002.

[44] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the SIGPLAN 1996 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, 1996.

[45] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176. ACM Press, 2004.

[46] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 336–349. ACM Press, 2003.

[47] W. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 140–145. ACM Press, 1999.

[48] J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–815, July 1992.

[49] J. E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, 1989.

[50] P. Song. UltraSPARC-III, aims at MP servers. In *Microprocessor Forum*, pages 29–34, 1997.

[51] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional verification of a multiple-issue, out-of-order, superscalar, Alpha processor–the DEC Alpha 21264 microprocessor. In *Proceedings of the 35th Annual Conference on Design Automation*, pages 638–643, 1998.

[52] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the SIGPLAN 2003 Conference on Compiler and Architecture Support for Embedded Systems*, pages 276–286, October 2003.

[53] W. Wang and J. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3):222–241, 1991.

[54] W. Wolf. *Computers as Components: Principles of Embedded Computing Systems Design.* Morgan Kaufmann, 2001.

[55] Y. Wu, R. Rakvic, L. Chen, C. Miao, G. Chrysos, and J. Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *MICRO35*, pages 134–145. IEEE Computer Society Press, 2002.

[56] Z. Wu and W. Wolf. Iterative cache simulation of embedded CPUs with trace stripping. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pages 95–99. ACM Press, 1999.

[57] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. In *ACM Computer Architecture News*, volume 23, pages 20–24, 1995.

[58] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. Vista: a system for interactive code improvement. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 155–164, 2002.