Automatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems

Santosh G. Abraham and Scott A. Mahlke
Hewlett-Packard Laboratories
Palo Alto, CA 94304
{abraham,mahlke}@hpl.hp.com

Abstract

Automation is the key to the design of future embedded systems as it permits application-specific customization while keeping design costs low. A key problem faced by automatic design systems is evaluating the performance of the vast number of alternative designs in a timely manner. For this paper, we focus on an embedded system consisting of the following components: a VLIW processor, instruction cache, data cache, and second-level unified cache. A hierarchical approach of partitioning the system into its constituent components and evaluating each component individually is utilized. The performance of each processor is evaluated independent of its memory hierarchy, and each of the caches is simulated using the traces from a single reference processor. Since the changes in the processor architecture do indeed affect the address traces and thus the performance of the memory hierarchy, the overall performance is inaccurate. To overcome this error, the changes in the processor architecture are modeled as a dilation of the reference processor's address trace, where each instruction block in the trace is conceptually stretched out by the dilation coefficient. This approach provides a projected cache performance that more accurately accounts for changes in the processor architecture. In order to understand the accuracy of the dilation model, we separate the possible errors that the model introduces and quantify these errors on a set of benchmarks. The results show the dilation model is effective for most of the design space and facilitates efficient automatic design.

1 Introduction

A wide range of devices and appliances ranging from mobile phones, printers, and cars have embedded computer systems. The number of embedded computers in these appliances far exceeds the number of general-purpose computer systems such as PCs or servers. In the future, the revenue stream from these embedded computers is expected to exceed those from general-purpose systems.

The design process for embedded computers is different from that of general-purpose systems. There is greater freedom in designing embedded computer systems because there is often little need to adhere to standards in order to run a large body of existing software. Since embedded computers are used in specific settings, they may be tuned to a much greater degree for certain applications. On the other hand, the revenue stream from a particular embedded system design is typically not sufficient to support a custom design. Though there is greater freedom to customize and the benefits of customization are large, the design budget available for customization is limited. Therefore, automated design tools are essential to capture the benefits of customization while keeping design costs down.

In this work, our design space consists of a VLIW processor and its associated memory hierarchy, consisting of Level-1 instruction, Level-1 data and Level-2 unified caches. The number and type of functional units in the VLIW processor may be varied to suit the application. The size of each of the register files may also be varied. Many other aspects of the VLIW processor, such as whether it supports speculation or predication, may also be changed. For each of the caches, the cache size, associativity, line size, and number of ports may be varied. Given this design space, an application, and its associated data sets, the objective is to determine a set of cost-performance optimal processors and systems. A given design is cost-performance optimal if there is no other design with higher performance and lower cost.

We focus on the performance evaluation of the memory hierarchy. The major problem is that it is not feasible to simulate all possible combinations of processor and cache from the specified space. Because of the multi-dimensional design space, the total number of possible designs can be exceedingly large. Even allowing a few of the VLIW processor parameters to vary easily leads to a set of 40 or more VLIW processor designs. Similarly, there may be 20 or more possible cache designs for each of the three cache types.

Evaluating a particular cache design for a particular VLIW design requires generating the address trace for that VLIW design and running this trace through a cache simulator. For a test program, such as the Postscript preview tool *ghostscript*, the sizes of the data, instruction, unified traces are 450M, 1200M, and 1650M respectively and the combined address trace generation and simulation process takes 2, 5, or 7 hours respectively. In a design space with 40 VLIW processors and 20 caches of each type, each cache has to be evaluated with the address trace produced by each of the 40 VLIW processors. Thus, evaluating all possible combinations of VLIW processors and caches takes (40x20x(2+5+7) hours which comes out to 466 days of computation. Such an evaluation strategy is clearly unacceptable.

We employ two complementary approaches to reduce the computation effort required to evaluate all the points in the design space. Firstly, we use the capabilities of a single-pass cache simulator to simulate multiple cache configurations in a single simulation run provided all the cache configurations have the same line size. Using this approach, the number of simulations is reduced from the total number of caches in the design space to the number of distinct cache line sizes in the design space. Thus, if all 20 caches in the design space have only one of two distinct line sizes, the overall computation effort is reduced by an order of magnitude.

Secondly, we use a hierarchical evaluation strategy; a common approach for evaluating complex systems with large design spaces. The complete system is divided into subsystems so that there is little coupling between subsystems. Each subsystem is individually evaluated in its own design space. The complete system is evaluated by combining the results of the evaluation of the individual subsystems and accounting for the effects of the (minimal) coupling between subsystems.

In our context, the overall system naturally separates into the VLIW processor, instruction cache, data cache, and unified cache subsystems. The overall execution time consists of the processor cycles and the stall cycles from each of the caches. We independently determine the processor cycles for a VLIW processor and the stall cycles for each cache configuration. Combined simulation of a VLIW processor and its cache configuration can take into account coupling between them, and produce more accurate results. For instance, sometimes processor execution may be overlapped with cache miss latencies, and the total cycles are less than the sum of the processor cycles and stall cycles. Accounting for such overlaps leads to more accurate evaluation. But, the simulation time required for doing cycle-accurate simulation is so large that we will not be able to examine a large design space using such accurate simulation techniques. We are more concerned with exploring a large design space than with the accuracy of the evaluation at each design point. Once we have narrowed down our choices to a few designs, the accurate evaluations can be done on each of the designs in this smaller set.

Using the hierarchical evaluation approach, we define a single reference VLIW processor and evaluate the cache subsystems only using the traces produced by a reference processor. Since the address trace generation and cache simulation are only performed using the reference processor, the total evaluation time is reduced by a factor equal to the number of VLIW processors in the design space (40 in our example). But, the VLIW processor design does indeed affect the address trace and hence influences the cache behavior. Therefore, using the cache stalls produced with the reference trace in the evaluation of a system with a non-reference VLIW processor will often lead to significant inaccuracies in evaluation. To overcome this problem, we measure certain characteristics of the object code produced for the non-reference processor with respect to that for the reference processor. In particular, the ratio of the text sizes, referred to as the *dilation*, is measured. We use the dilation to adjust the cache misses and stalls for the non-reference processor to more accurately model the performance of the memory system.

Though we are primarily motivated by the design of embedded systems, this approach is useful even for evaluating general-purpose systems. Quite often, architectural or compiler techniques are evaluated solely at the processor level without quantifying their impact on memory hierarchy performance. For instance, code specialization techniques, such as inlining or loop unrolling may improve processor performance, but at the expense of instruction cache performance. The evaluation approach described in this paper can also be used in these situations to quantify the impact on memory hierarchy performance in a simulation-efficient manner.

2 Related Work

One important area of previous research is work on automatic design of embedded systems. The automatic synthesis of transport-triggered architectures within the MOVE framework has been investigated [1]. A template-based processor design space is automatically searched to identify a set of best solutions. In the SCARCE project, a framework for the design of retargetable, application-specific VLIW processors is developed [2]. This framework provides the tools to tradeoff architecture organization and compiler complexity. A hierarchical approach is proposed for the design of systems consisting of processor cores and instruction/data caches [3]. A minimal area system that satisfies the performance characteristics of a set of applications is synthesized. In contrast to previous work, our framework permits us to explore a large parameterized processor design space in conjunction with a parameterized memory hierarchy design space.

A second area of previous research focuses on the development of memory hierarchy performance models. Cache models generally assume a fixed trace and predict the performance of this trace on a range of possible cache configurations. In a typical application of the model, we derive a few trace parameters from the trace and use them to estimate misses on a range of cache configurations. For instance, models for fully-associative caches employ an exponential or power function model for the change in working set over time. These models have been extended to account for a range of line sizes[4, 5]. Other models have been developed for direct-mapped caches [6], instruction caches [7] [8], multi-level memory hierarchies [9], and multiprocessor caches [10].

The analytic cache model by Agarwal et al, referred to subsequently as the AHH model, estimates the miss rate of set-associative caches using a small set of trace parameters derived from the address trace [11]. The AHH model has been validated by determining the effectiveness of the model in predicting the cache miss rates of a range of caches over a range of benchmarks. The mean percentage error in miss rate for direct-mapped caches with a block size of 4 bytes is 4% but the mean error increases to 22% for set-associative caches with a line size of 16 bytes. In general, the accuracy decreases as the line size increases.

Instead of using cache models to estimate the performance of various caches on a fixed trace, we use cache models to estimate the performance of caches on dilated versions of a reference trace. We do not use the AHH model to completely eliminate simulation runs because the accuracy of the AHH model by itself is not adequate. Instead, we use the AHH model to interpolate/extrapolate the results from actual simulation runs on the reference trace to the performance of caches on dilated traces. Steenkiste [12] examined the effect of code density on the instruction cache performance of RISC processors. We also model the effect of dilation on instruction cache performance as a decrease in the effective line size. The effective line size is usually not simulatable because it is not a power-of-two and therefore interpolation is usually required. As opposed to a curve-fitting approach for interpolation [12], we use the AHH model to accurately estimate the misses of an unrealizable cache by interpolating between the misses of the closest two power-of-two line size caches. Prior work was limited to instruction cache performance. We develop models for estimating unified cache

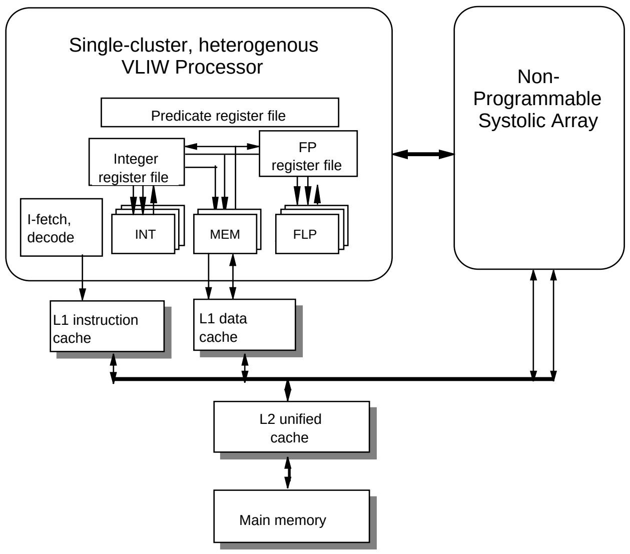

Figure 1: Overall design space

performance for dilated traces and examine the effect of dilation on unified cache performance.

3 Design and Evaluation System

This section describes our tool chain to design and evaluate a wide range of VLIW processors. In conventional systems, each of the separate modules of the chain is designed and developed for a particular processor architecture. The functions of some other modules, such as instruction format design, are done manually. The unique aspect of our tools is that they work automatically for any member of the parameterized design space. We first present the overall design space including that of the memory hierarchy. A brief overview of the design system is then given. Last, the particular aspect of the system focused on in this paper, the memory hierarchy evaluation system, is described.

3.1 Design space

The overall design space targeted by our system is shown in Figure 1. The design space consists of a single-cluster VLIW processor and an optional hardware accelerator in the form of a non-programmable systolic array. The VLIW processor is parameterized by each of its components, including number and type of function units, size of the register files, whether the processor supports predication or speculation. Design parameters are carefully chosen to deliver a desired level of performance or cost.

The memory system of the design is composed of a Level-1 data cache, Level-1 instruction cache, and a Level-2 unified cache. Each of the caches comprising the memory system is also parameterized with respect to cache size, associativity, line size, and number of ports. We require that the parameters chosen for the memory system are such that inclusion is satisfied between the data/instruction caches and the unified cache. The inclusion property states that the unified cache contains all items contained in the data/instruction caches and is enforced in the majority of systems. This property decouples the behavior of the unified

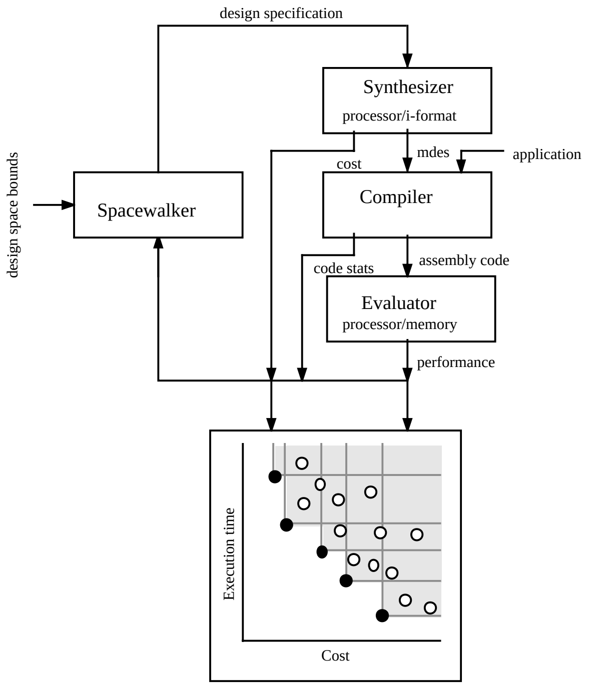

Figure 2: High level view of automatic design system

cache from the data/instruction caches in the sense that the unified cache misses will not be affected by the presence of the data/instruction caches. Therefore, the unified cache misses may be obtained independently, regardless of the configuration of the Level-1 caches, by simulating the entire address trace.

3.2 Automatic design system

The overall design system is an iterative system that determines cost-performance optimal designs within a user-specified range. An abstract view of the system is presented in Figure 2. The driver of the system is a module referred to as the *spacewalker*. The spacewalker is responsible for defining high-level specifications for candidate designs to be investigated. The spacewalker derives the cost and performance for each candidate design using three subsystems: synthesizer, compiler, and evaluator. The synthesis system creates the design for the processor, instruction format, memory hierarchy, and optional hardware accelerator from a high-level input specification [13]. From the design, the system cost can be readily determined. In addition, the synthesis system creates a machine-description file (mdes) to describe the relevant parts of the system to the compiler.

The compiler is responsible for mapping the application to assembly code for the synthesized processor. The compiler is an extended version of the Trimaran compiler infrastructure [14]. The Trimaran infrastructure is a flexible compiler system capable of generating highly optimized code for a family of VLIW processors. The compiler is retargetable to all processors in the design space and utilizes the machine description to generate the proper code for the synthesized processor. The last component of the overall design system is a set of performance evaluators. Performance of the processor, hardware accelerator, and memory hierarchy are separately evaluated; then, they are combined to derive the overall system performance. Performance of the processor and accelerator are estimated using schedule lengths and profile statistics. The memory system performance is derived using a combination of trace-driven simulation and performance estimation described in the next section.

Each design is plotted on a cost/performance graph as shown in Figure 2. The sets of points that are minimum cost at a particular performance level identify the set of best designs or the *pareto curve*. After the process is completed for one design, the spacewalker creates a new design and everything is repeated. The spacewalker uses cost and performance statistics of the previous design as well as characteristics of the application to identify a new design that is likely to be profitable. The spacewalking process terminates when there are no more likely profitable designs to investigate.

3.3 Memory system evaluation

The number of designs that need to be explored by spacewalker is large even when sophisticated search heuristics are used. Hence, it is necessary to develop highly efficient performance evaluation tools to make this approach feasible. For this work, we focus on producing cache performance metrics for any design point in a simulation-efficient manner. Two techniques are used to accomplish this objective. First, a retargetable memory simulation system is needed to simulate arbitrary design points in the space. These points are referred to as reference processors. Second, a performance modeling system is used to estimate the performance of other design points or non-reference processors. The remainder of this section focuses on a description of the retargetable memory simulation system. The next section describes the performance modeling technique.

The organization of the retargetable memory simulation system is presented in Figure 3. The central components are: the assembler, linker, emulator, execution engine, trace generator, and cache simulator. The input to the system is the scheduled and register allocated assembly code produced by the compiler for the target processor. The input is used in two parallel paths. The assembler and linker create a binary representation of the application for the target processor. The emulator and execution engine converts the input code to an instrumented executable program for an existing platform. Execution of the instrumented program produces a trace, referred to as an *event trace*. The trace generator combines the event trace and the binary for the target processor to produce an address trace to drive a cache simulator. The output of the system is the number of misses for the particular cache configuration that is evaluated.

Assembler. The assembler maps the scheduled code into a machine-dependent binary representation specified by the instruction format. A customized instruction format is co-synthesized with each VLIW processor. The instruction format specifies all of the available binary instruction encodings for the processor. Our instruction format system generates variable-length, multi-template formats to facilitate reducing overall code size [15]. The assembler examines each set of operations that are concurrently scheduled and selects the best template to encode the operations in a single instruction. The assembler uses a greedy template selection heuristic based on two criteria to minimize code size. First, the template that requires the fewest bits is preferred. Second, the template should have sufficient multi-no-op bits to encode any no-op instructions that follow the current instruction.

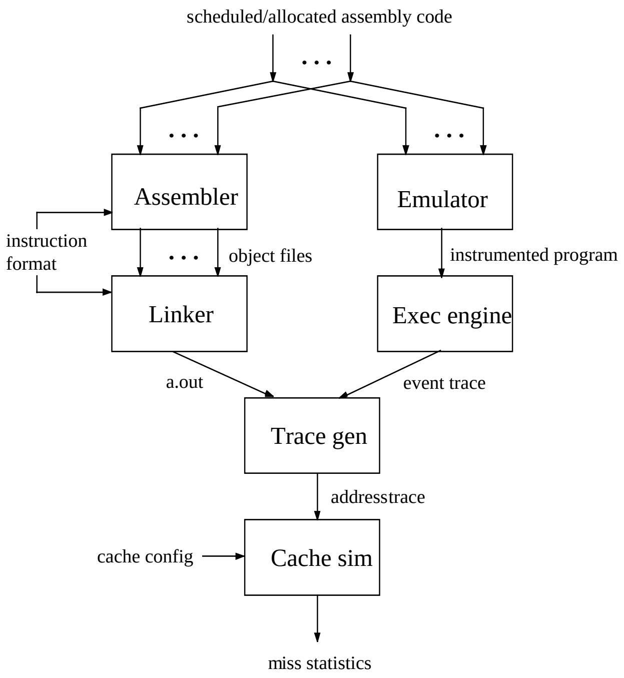

Figure 3: Overview of the memory simulation system

After a template is selected, the assembler fills in the template bits with the appropriate values for the current instruction. The final output of the assembler is a binary representation for each procedure in the original application known as a relocatable object file.

Linker. The linker combines all the object files for the individual functions of the application into a single executable file. In this process, the linker is responsible for code layout, instruction alignment, and assigning the final addresses to all of the instructions. In the current system, the compiler handles intra-procedural code layout, while the linker is responsible for inter-procedural layout. Branch profile information is used in both phases to place blocks of instructions or entire functions that frequently execute in sequence near each other. The goal is to increase spatial locality and instruction cache performance. Instruction alignment rules are derived from the instruction format and fetch configuration of the synthesized processor. Instructions that are branch targets are aligned on *packet boundaries* (a packet consists of the set of bits fetched from the instruction cache in a single cycle) to avoid instruction cache fetch stalls for branch targets at the expense of slightly larger code size. The last step is to assign addresses to all instructions in the executable file.

Emulator and execution engine. The dynamic behavior of the application is captured using a combination of an emulator and an execution engine. The emulator converts the assembly code for the synthesized processor into an equivalent assembly code for an existing platform, such as a HP workstation. Essentially, the emulator is an assembly code translation tool. In addition, the emulator instruments the output assembly code to record important dynamic events for memory system evaluation.

Symbol	Description	Symbol	Description
P_{ref}	Reference processor	d	Dilation with respect to T_{ref}
P_i	Arbitrary processor	$M(IC_j,P_i,d)$	Cache misses on IC_j
T_{ref}	Reference processor trace	τ	Number of references per granule
T_i	Arbitrary processor trace	N	Number of granules in trace
C	Cache	$u(L)$	Average unique cache lines in granule
$C(S,A,L)$	Cache with S sets, associativity A, line size L	$u(1)$	Average unique words in granule
S	Number of sets in cache	$P(L,a)$	Probability of a blocks in set
A	Cache associativity	p_1	Average isolated references per granule
L	Cache line size	l_{av}	Average run length
IC_j	Instruction cache j	$Coll(S,A,L)$	Collisions in $C(S,A,L)$
DC_j	Data cache j	UC_j	Unified cache j

Table 1: Description of symbols

These include procedures entered/exited, basic blocks entered, branch directions, and load/store data addresses. For our system, the IMPACT emulation tools for the HP PA-RISC architecture are utilized [16]. The instrumented assembly code is compiled using the host system assembler and linker to produce an instrumented-executable version of the application. The execution engine then runs the program on the host system to produce an event trace that records the dynamic program behavior as a high level sequence of tokens. Note that the event trace is dependent on the scheduled/allocated assembly code produced for the synthesized processor, but it is independent of the instruction format and organization of the executable file for the synthesized processor.

Trace generator. The trace generator creates an instruction and/or data address trace that models the application executing on the synthesized processor. This is accomplished by symbolically executing the synthesized processor executable file. Symbolic execution is driven by the event trace produced by the execution engine. The event trace identifies the dynamic instructions that must be executed by providing control flow events (e.g., enter a basic block, direction of a branch, or predicate value). The trace generator just maps control flow events to the appropriate the sequence of instruction addresses obtained from the executable file that are visited to create the instruction trace. The event trace also provides data addresses accessed by load and store operations. The trace generator simply passes these addresses through at the time when the load/store is executed to create the data trace. The trace generator is configurable to create instruction, data, or joint instruction/data traces as needed.

Cache simulator. The cache simulator used in our system is the Cheetah simulator [17]. Cheetah is capable of simulating a large range of caches of different sizes and associativities in a single pass. The line size is the only parameter that is fixed. Cheetah also uses sophisticated inclusion properties between caches to reduce the amount of state that must be updated, thereby reducing the overall simulation time. In our usage, all caches in the design space for each synthesized processor is simulated and the number of misses for each cache is tabulated. Separate runs are needed for each line size that is considered. Also, separate runs are required for each of the caches (Level-1 instruction, Level-1 data, and Level-2 unified) as each requires a different address stream.

4 Dilation Model

In this section, we describe the processes and models associated with estimating the cache misses on the trace produced by an arbitrary VLIW processor. The misses produced by an arbitrary VLIW processor are estimated using (1) the cache misses obtained through simulation on the reference VLIW processor trace and (2) models relating the cache behavior of two traces. As we described earlier, we can only afford to perform complete simulations on the reference trace.

4.1 Process

In the following, we assume a fixed application running on a fixed data set. Let P_{ref} be the reference VLIW processor and P_i be an arbitrary VLIW processor from the design space. In our experiments, we use a narrow-issue processor for P_{ref} and a comparatively wide-issue processor for P_i. Let T_{ref} be the trace produced by P_{ref} and T_i the trace produced by P_i. Trace modeling parameters, *TP*, are a small number of parameters derived from the trace, T_{ref}. Let IC_j, DC_j, and UC_j represent instruction, data and unified cache configurations. Let $C(S,A,L)$ represent a cache with S sets, associativity A, and a line size of L. When the specific cache parameters are not relevant, we abbreviate $C(S,A,L)$ to C. Let $M(IC_j,P_i)$ be the true misses produced by the trace, T_i, on the instruction cache, IC_j. Since we do not simulate using T_i, we estimate $M(IC_j,P_i)$ using the trace modeling parameters, *TP*, and misses, $M(IC_k,P_{ref})$ on some arbitrary set of feasible IC_k. In our context, a cache is feasible if its line size and number of sets are powers of two, and its associativity is an integer. The notation introduced in this section is summarized in Table 1.

The process is best understood as a series of three steps each associated with an approximation and hence an attendant inaccuracy. In the experiment section, we determine the extent of these inaccuracies on particular benchmarks and machine configurations. We first state the assumption associated with each step (in italics) and then discuss the implications of the assumption and the rationale for making the assumption.

1. *The data trace component of T_i is identical to the data trace component of T_{ref}. The instruction trace component of T_i contains the same sequence of basic blocks as that of T_{ref}, except*

that the sizes of the individual basic blocks are typically larger in T_i.

We require that P_{ref} and P_i have the same data speculation and predication features, because these features have a large impact on address traces. When the design space covers machines with differing predication/speculation features, we use several P_{ref} processors, one for each unique combination of predication and speculation. Given that the data trace components are identical in T_i and T_{ref},

$$M\left(DC_j,P_i\right) \approx M\left(DC_j,P_{ref}\right) \tag{4.1}$$

Processor P_i may have a different data trace on account of two factors. Firstly, more of the operations in P_i may be speculated and therefore, the data trace may contain more of these speculated load addresses. However, the compiler does use heuristics to speculate the most profitable operations. Therefore, the number of spurious load addresses is not expected to be large. Secondly, P_i may have larger (or smaller) amount of spills due to register pressure, depending on the relative size of the register files, the relative issue-width of the processor, and the amount of instruction-level parallelism. Even if register spill code introduces additional loads/stores, these are likely to have high locality and not increase the number of data cache misses significantly.

We assume that the basic block traces of the two processors are identical, i.e., the sequence of basic blocks in each of the two traces is identical. The compiler may perform different optimizations depending on the machine widths and associated features. But, in our compiler, these optimizations are limited to optimizations within superblocks/hyperblocks. The generation of superblocks/hyperblocks is not machine-dependent currently. Since the optimizations that can affect the basic block trace are machine-independent, e.g. the degree of loop unrolling, the basic block trace is identical for P_{ref} and P_i.

Though the basic block traces for P_{ref} and P_i are identical, the size of each basic block in P_i differs from that in P_{ref}. This occurs because the wider-issue processor has a wider instruction format. The wider instruction format of P_i is generally more inefficient and will require more bits to represent the same set of operations. The operand formats of the wider processor are also typically larger due to larger register files. Additionally, the wider-issue processor tends to speculate more often, thereby increasing both the dynamic size of the address trace as well as the static code size. Let the *dilation* of a basic block be the ratio of the size of a basic block in to P_i that in P_{ref} and *text dilation d* be the ratio of the overall text size of the benchmark in P_i to that in P_{ref}.

2. *The dilation of all basic blocks is uniform and equal to the text dilation d.*

A trace, dilated by *d*, is derived from T_{ref} as follows. The length of each basic block in T_{ref} is increased by a multiplicative factor *d*. Additionally, the starting address of each basic block is adjusted to ensure that the dilated basic blocks do not overlap in the dilated trace. Let $M(IC_j ,P_i ,d)$ represent the instruction cache misses on this dilated trace and let $M(UC_j ,P_i ,d)$ represent the unified cache misses on a trace where the instruction component is dilated as described. Under the uniform dilation assumption, except for minor differences in the positioning of basic blocks in the address space, this dilated trace is identical to T_i. Therefore,

$$M\left(IC_j,P_i\right) \approx M\left(IC_j,P_{ref},d\right) \tag{4.2}$$

and

$$M\left(UC_j,P_i\right) \approx M\left(UC_j,P_{ref},d\right) \tag{4.3}$$

The assumption of uniform dilation implies that all basic blocks are increased in size by the same amount on the wider-issue processor. In general, the basic block contents do indeed vary on wider processors due to different scheduling decisions. However, the dominant factor in the code size increase is the wider instruction format. With relatively fixed basic block contents, each basic block will increase in size proportional to the width that the average instruction increases, which is equivalent to the overall text dilation. In the experiment section, we examine the variability of dilation across blocks and the degree to which the text dilation represents the true dilation.

3. *$M(IC_j ,P_{ref},d)$ can be estimated accurately from $M(IC_k ,P_{ref})$, for some set of IC_k and the trace parameters, TP. Similarly, $M(UC_j ,P_{ref} ,d)$ can be estimated accurately from $M(UC_k ,P_{ref})$ for some set of UC_k and trace parameters, TP.*

This step is again associated with some inaccuracies that may result in the estimated misses differing from the actual misses on the dilated trace of P_{ref}. The trace model we use may not capture the behavior of the reference trace in sufficient detail. Also, the manner in which we use the trace model to obtain the miss behavior of the dilated reference trace may not be sufficiently accurate. However, the baseline AHH model has been shown to be effective in predicting cache miss rates for a large range of caches. Further, the differences in the dilated and reference traces can be intuitively characterized allowing the application of the AHH model to be extended in a straight-forward manner. The derivation of the formulas to approximate $M(IC_j ,P_{ref} ,d)$ and $M(UC_j ,P_{ref},d)$ are presented in the remainder of this section.

4.2 Trace model

In this subsection, we review the AHH cache model [11]. In the next subsection, we use the trace parameters of the AHH model to estimate misses of the dilated trace. The AHH model is motivated by the need to obtain quick estimates on cache performance for a wide range of set-associative cache configurations, without resorting to time-consuming and/or expensive trace-driven simulation or hardware measurement. A few parameters are derived from a single-pass through the address trace. Analytic models relate these parameters to miss rates of arbitrary cache configurations.

The AHH model divides the trace into *N* time granules, each containing a certain number of references, τ. Within each granule, we sort the references in each granule based on the address values, so that addresses that belong to a run will appear consecutively. For our work, all addresses are word addresses. An address is either part of a run, i.e. there are other references in the granule that neighbor this address, or the address is an isolated (singular) address. Let *u(1)*, be the average number of unique references in a granule. Let p_1 be the average fraction of isolated references in a granule, i.e. the average of the ratios of isolated references to unique references over all granules. Let l_{av} be the average run length, the number of consecutive addresses composing each run averaged over all the runs in a granule and over all the granules.

These three basic parameters, viz. *u(1)*, p_1, and l_{av}, are used to derive a set of secondary parameters, viz. p_2, *u(L)*, and *P(L,a)*. The parameter p_2 corresponds to the state-transition probability of transferring from the current run of sequential addresses to a new run; *u(L)* is the average number of unique cache lines accessed in a time granule; and *P(L,a)* is the probability that *a* cache lines of size *L* are mapped into a set.

$$p_2 = \frac{l_{av} - (1 + p_1)}{l_{av} - 1} \qquad (4.4)$$

$$u(L) = u(1)\frac{1 + p_1/L - p_2}{1 + p_1 - p_2} \qquad (4.5)$$

$$P(L,a) = \binom{u(L)}{a}\left(\frac{1}{S}\right)^a\left(1 - \frac{1}{S}\right)^{u(L)-a} \qquad (4.6)$$

The AHH model characterizes cache misses into start-up, non-stationary and intrinsic interference misses. We assume that steady-state interference misses dominate and ignore the start-up and nonstationary misses. We are primarily interested in using an available miss rate of a cache, $C_1(S_1, A_1, L_1)$ to estimate the miss rates of another cache, $C_2(S_2, A_2, L_2)$. The steady state miss rate, $m(C_2)$, is related to $m(C_1)$ by

$$m(C_2) = \frac{Coll(S_2,A_2,L_2)}{Coll(S_1,A_1,L_1)} m(C_1) \qquad (4.7)$$

where

$$Coll(S,A,L) = u(L) - \sum_{a=0}^{a=A} S \cdot a \cdot P(L,a) \qquad (4.8)$$

Thus, given the three basic parameters, $u(1)$, p_1, l_{av}, and the miss rate for any cache, we can estimate the miss rate of any other cache.

4.3 Estimating performance of dilated traces

In this section, we describe the utilization of the AHH model to estimate the instruction and unified cache performance of dilated traces. In each case, we first determine the appropriate values for the basic parameters of the AHH model, $u(1)$, p_1, l_{av}. Subsequently, we use these parameters and the misses on the reference traces to determine the instruction and unified cache misses on dilated traces.

In the case of the instruction cache, we are only interested in the instruction component of the trace. Therefore, in determining the basic parameters, $u(1)$, p_1, l_{av}, we filter out the data component and divide the instruction component into granules. We process each granule as described earlier and obtain values for the three basic parameters, $u(1)$, p_1, l_{av}, for the entire trace. In the case of the unified cache, we have to separate out the instruction and data components of the trace because only the instruction component is dilated. Therefore, we derive a separate set of parameters for the instruction component and the data component. We divide the unified trace into fixed-size granules and then separately sort the instruction and data addresses. For each of the two components, we obtain values for the three basic parameters. Thus, we obtain $u_I(1)$, p_{1I}, and l_{avI} for the instruction component and $u_D(1)$, p_{1D}, and l_{avD} for the data component. For a specific cache configuration,

$$u(L) = u_I(L) + u_D(L) \qquad (4.9)$$

where $u_I(L)$ is a function of the three parameters obtained for the instruction component of the trace and $u_D(L)$ is a function of the parameters for the data component. Once we obtain $u(L)$, we use equations (4.6) and (4.8) to determine collisions for a particular cache configuration as in the instruction cache case.

4.3.1 Instruction cache performance

First, consider estimating the instruction cache misses, $M(IC_j, P_{ref}, d)$. Dilating the trace by *d* is equivalent to contracting the line size of IC_j by *d* and leaving the other parameters of the cache, viz. number of sets and associativity, unaltered. Thus, a trace dilation of two is equivalent to reducing the line size from, say, 16 bytes to 8 bytes. However, a cache with line size *L / d* may not be feasible in general. In this case, we choose two line sizes, $L_l = 2^l$ and $L_u = 2^{l+1}$ for integer *l*, such that $L_l < L / d < L_u$. We estimate $M(IC(S,A,L / d),P_{ref})$ by interpolating between $M(IC(S,A,L_u),P_{ref})$ and $M(IC(S,A,L_l),P_{ref})$. A linear interpolation is not suitable because the misses are a very nonlinear function of line size. We use the AHH trace parameters and model to generate the more sophisticated interpolation as described below.

$$M(IC(S,A,L)) = \frac{M(IC(S,A,L_l)) - M(IC(S,A,L_u))}{Coll(IC(S,A,L_l)) - Coll(IC(S,A,L_u))} Coll(IC(S,A,L)) + \frac{\left[\begin{array}{l} M(IC(S,A,L_u)) \cdot Coll(IC(S,A,L_l)) \\ - M(IC(S,A,L_l)) \cdot Coll(IC(S,A,L_u)) \end{array}\right]}{Coll(IC(S,A,L_l)) - Coll(IC(S,A,L_u))} \qquad (4.10)$$

Note that at the two end points $IC(S,A,L_l)$ and $IC(S,A,L_u)$, the right hand side of (4.10) evaluates to $M(IC(S,A,L_l))$ and $M(IC(S,A,L_u))$ respectively.

Thus to determine $M(IC(S,A,L),P_{ref},d)$, we first transform this problem into determining the misses on *IC(S,A,L / d)* using the reference trace, $M(IC(S,A,L / d),P_{ref})$. In the general case, *L / d* is not a power of two and therefore not a feasible line size. We use Equation (4.10) above to estimate $M(IC(S,A,L / d),P_{ref})$ from $M(IC(S,A,L_l),P_{ref})$ and $M(IC(S,A,L_u),P_{ref})$, where L_l and L_u are the immediately lower and higher line sizes that are powers of two.

4.3.2 Unified cache performance

Consider estimating the unified cache misses, $M(UC(S,A,L),P_{ref},d)$ on a reference trace dilated by *d*. In the case of the instruction cache, we were able to transform the problem into one of determining the misses on a related cache configuration using the undilated trace. This approach is not feasible for the unified cache because of the mix of an undilated data component with a dilated instruction component.

As described in Section 4.2, we derive the following basic parameters from a simulation-like run through the unified address trace: $u_D(1)$ and $u_I(1)$, the average number of unique data and instruction references in a granule, p_{1D} and p_{1I}, the average probability of a singular reference in the data and instruction components, and l_{avD} and l_{avI}, the average run length on the data and instruction components. From these basic parameters and equations (4.4), (4.5), (4.9), we derive *u(L)*, for a specific line size, *L*.

Let $Coll(T_{Pref}, UC(S,A,L))$ and $Coll(T_{Pref,d}, UC(S,A,L))$ represent the collisions in a unified cache, with and without dilation. We derive $Coll(T_{Pref}, UC(S,A,L))$ from (4.6) and (4.8). The procedure for determining $Coll(T_{Pref,d}, UC(S,A,L))$ takes into account that the instruction stream is dilated but not the data stream. In estimating the instruction cache misses, we transformed the dilation of the instruction stream to an equivalent reduction in line size. In a similar manner, we approximate $u(L,d) \approx u_D(L) + u_I(L / d)$ We then substitute $u(L,d)$ in the following, which are modified versions of (4.6) and (4.8) respectively, that account for dilation of just the instruction component of the trace.

$$P(L,a,d) = \binom{u(L,d)}{a}\left(\frac{1}{S}\right)^a\left(1-\frac{1}{S}\right)^{u(L,d)-a} \tag{4.11}$$

$$Coll\left(T_{ref,d}, C(S,A,L)\right) = u(L,d) - \sum_{a=0}^{a=A} S \cdot a \cdot P(L,a,d) \tag{4.12}$$

Now that we have the two collision terms and the misses on the reference trace using simulation, we can then estimate misses of the dilated trace using a modified version of (4.7).

$$M\left(UC(S,A,L), P_{ref}, d\right) = \frac{Coll\left(T_{\Pr ef,d}, UC(S,A,L)\right)}{Coll\left(T_{\Pr ef}, UC(S,A,L)\right)} M\left(UC(S,A,L)\right) \tag{4.13}$$

where we obtain $M(UC(S,A,L))$ through simulation.

5 Validation and Evaluation

In this section, we validate and evaluate the dilation model and its implementation. We quantify the inaccuracies introduced by the two major steps in our approach, viz. (1) the assumption that all blocks are uniformly dilated by the text dilation and (2) the estimation of the cache misses of the dilated trace using the cache simulation results on the reference trace and the AHH trace parameters.

Our choice of benchmark applications is influenced by two major factors. Firstly, the overall focus of our system is in automatically generating embedded systems on a chip tuned for specific applications. We are therefore interested in multimedia-intensive benchmarks that are likely to be targeted by embedded systems. We use a subset of MediaBench, a set of benchmarks developed by Mangione-Smith and his group for facilitating work on embedded systems [18]. Secondly, since a major focus of this work is in correctly estimating instruction and unified cache behavior, we choose benchmarks where instruction and unified cache behavior have a significant effect on overall performance. In benchmarks where instruction cache miss ratios are small, it is not that important to account for variations in instruction traces due to changes in machine architecture. We chose the following benchmarks from MediaBench with the highest instruction cache miss rates: *epic*, *ghostscript*, *mipmap*, *pgpdecode*, *pgpencode*, *rasta*, *unepic*. We also chose three benchmarks from the SPEC suite because it is well understood by the microarchitecture research community. Again, benchmarks with higher relative instruction cache miss rates were chosen: *085.gcc* from the SPECINT-92 suite; *099.go* and *147.vortex* from the SPECINT-95 suite. We present results for all benchmarks in tabular form and show in depth analysis for one representative application, *085.gcc*.

In our experiments, we use a narrow 1111 VLIW processor with one each of integer, float, load/store and branch units as the reference processor. We also use the following processors as the target (arbitrary) processors: 2111, 3221, 4221, and 6332, where the digits denote the number of integer, float, memory, and branch units, respectively. Note that the reference processor can issue up to 4 operations per cycle and the 2111, 3221, 4221, and 6332 target processor can issue up to 5, 8, 9, and 14 operations per cycle, respectively. With the narrow VLIW processor as the reference processor, the experiments measure the effectiveness in estimating the cache miss behavior of the wider processors from that of the narrow processor.

Two cache configurations are used for the experiments. First, a small configuration consisting of a 1K, direct-mapped instruction cache with a line size of 32 bytes and a 16K, 2-way set associative unified cache with a line size of 64 bytes. Second, a large configuration consisting of a 16K, 2-way set associative instruction cache with a line size of 32 bytes and a 128K, 4-way set associative unified cache with a line size of 64 bytes.

5.1 Validation with Impact

The first step in the evaluation was to verify accuracy of our memory simulation system. The Impact compiler from the University of Illinois provided us with an alternative set of simulation tools [16]. Impact's cache simulator uses detailed modeling and accurate stall cycles for a superscalar processor. In order to test the correctness of our memory hierarchy evaluation modules, we determined the instruction and data cache miss rates for several benchmarks and a range of cache configurations using both sets of tools. There were some small differences in the number of misses produced by the two systems. These differences could largely be attributed to the more detailed simulation in Impact involving slightly different handling of writes and write-buffer issues. After accounting for these differences, the final miss rates generated by the two simulation systems were virtually identical.

5.2 Distribution of dilation

The dilation of a block is the ratio of its sizes on the wide and narrow reference processors. An important assumption in our model is that blocks are dilated by the same amount and this amount is the ratio the text size of the wide processor to the narrow processor.

The first row of graphs in Figure 4: plots the static and dynamic cumulative distributions of dilations of the arbitrary processors with respect to the reference processor. In this figure, the top left graph shows the results for *085.gcc* and the top right graph for *ghostscript*. The static distribution plots the fraction of blocks in the executable whose dilations are less than or equal to a threshold specified on the X-axis. The dynamic distribution plots the weighted fraction of blocks, whose dilations are less than or equal to a threshold, where the weighting of each block is equal to its dynamic execution frequency. There are a set of three static curves one for each of three arbitrary processors, viz. 2111, 3221 and 6332. In order to reduce clutter, the results for the 4221 processor are omitted. These two graphs also show a similar set of three dynamic curves. These graphs help us examine the validity of our assumption that all blocks are dilated uniformly by the text dilation. If that was indeed the case, the static and dynamic curves would each be a step function, being at zero for dilation values less than the text dilation and at one for dilations more than the text dilation. The steeper the curve is in rising from zero to one, the more accurate the uniform dilation assumption. Note that the assumption is generally more accurate for the 2111 processor than for the wider 6332 processor. Also, note that the dynamic distribution tracks the static distribution quite closely, but less so for the wider processors. This behavior indicates that the dilation of the more frequently executed blocks is similar to the dilation of other blocks. When the dynamic distribution is significantly different from the static distribution,

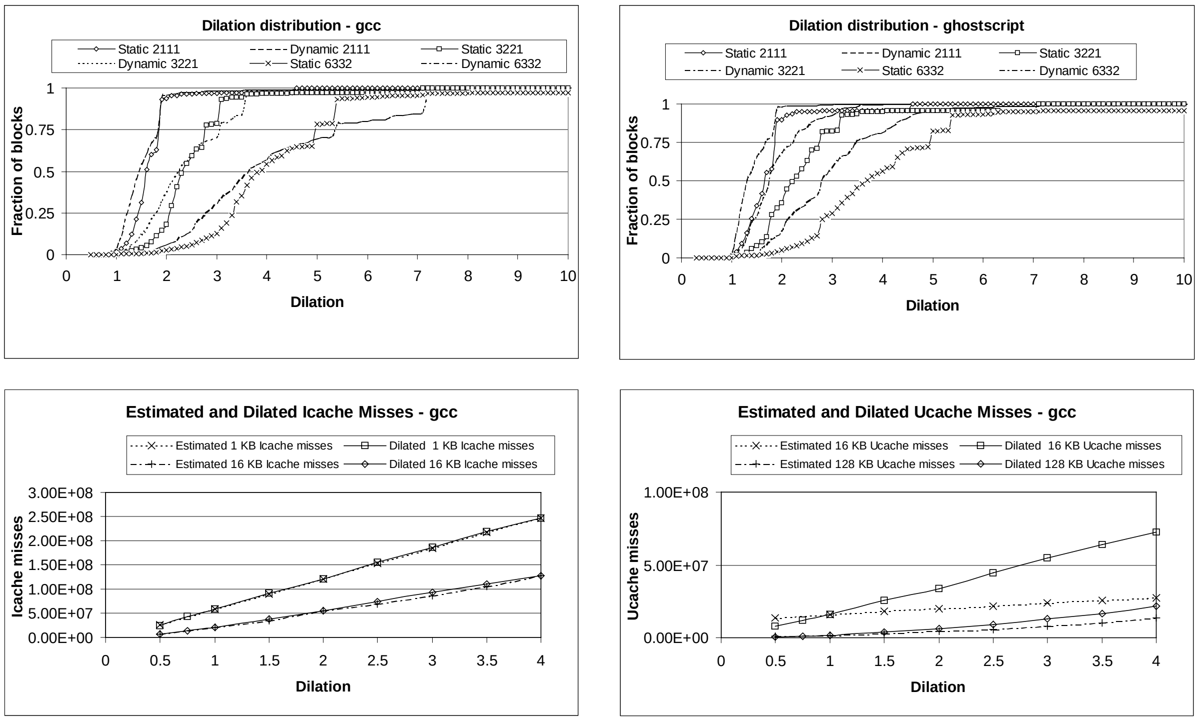

Figure 4: Dilation distribution and misses versus dilation

choosing the static text dilation for dilating the reference trace may not be accurate.

Table 2 shows the text dilation for all the benchmarks and machine architectures we used in the experiments. The text dilations typically fall in the middle of the range where the static and dynamic dilation distributions rise from 0 to 1. The positioning of the text dilation relative to the distribution curves justifies our use of the text dilation to approximate the dilations of individual blocks. Recall that the processors issue up to four, five, eight, nine and 14 operations per cycle. For all the benchmarks, the text dilation increases much more gradually than the issue width. For the 2111, 3221 and 4221 processors, the text dilation is less than 2.5, indicating that models that accurately estimate performance up to a dilation of 2.5 are sufficient for such machine architectures. It is only for the 6332 processor that text dilations are in the range 2.5 through 3.25.

5.3 Dilated versus estimated miss rates

In this section, we evaluate the accuracy of the second step in our approximation, viz., estimating the miss rates of the dilated trace from the miss rates on the reference trace. The bottom row of graphs in Figure 4: plots the misses on the dilated trace versus the dilation for the benchmark *085.gcc*. The bottom left graph

Benchmarks	**Text Dilation**				
	1111	**2111**	**3221**	**4221**	**6332**
085.gcc	1.00	1.40	1.99	2.28	3.24
099.go	1.00	1.40	1.89	2.17	3.08
147.vortex	1.00	1.36	1.74	2.00	2.78
epic	1.00	1.26	1.76	1.92	2.65
ghostscript	1.00	1.40	1.99	2.15	3.01
mipmap	1.00	1.32	1.78	2.51	2.81
pgpdecode	1.00	1.40	2.00	2.28	3.25
pgpencode	1.00	1.36	1.97	2.24	3.18
rasta	1.00	1.26	1.69	1.91	2.70
unepic	1.00	1.29	1.66	1.80	2.47

Table 2: Text dilation for all benchmarks

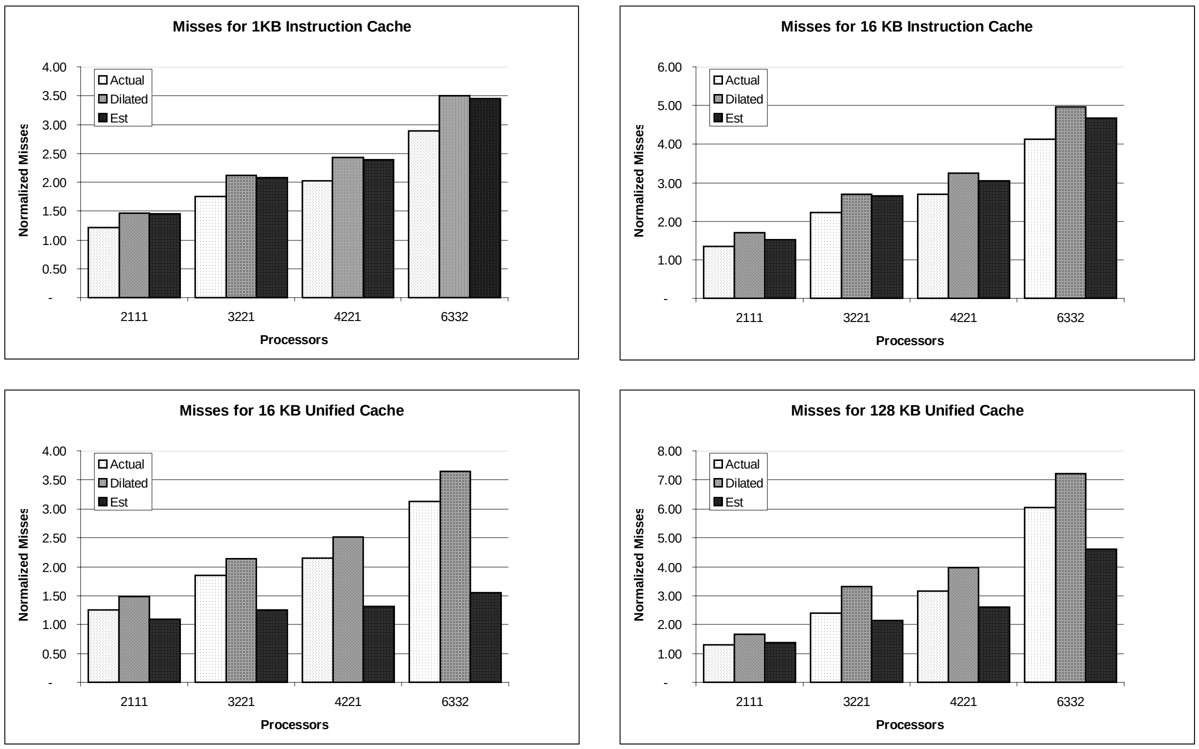

Figure 5: Actual, Dilated and Estimated Misses for 085.gcc

shows the instruction cache misses for a direct-mapped 1KB cache and a two-way set associative 16 KB cache, each with a line size of 32 bytes. The bottom right graph shows the unified cache misses for a two-way set-associative 16 KB unified cache and a four-way set-associative 128 KB unified cache, each with a line size of 64 bytes. These figures also plot the misses estimated by our dilation model for a range of dilations. The extent to which the estimated misses track the dilated misses shows the accuracy of our dilation model in estimating misses for dilated traces. For the instruction cache, the estimated misses track the dilated misses very closely throughout the entire range of dilations. For the large 128 KB unified cache, the estimated misses track the dilated misses very well. For the small 16 KB unified cache, the estimated misses tracks the dilated misses only up to a dilation of two. Recall the instruction cache miss modeling interpolates between the misses of two realizable caches, whereas the unified cache miss modeling extrapolated from the miss behavior of the cache on the reference trace. In general, the interpolation for the instruction cache is more accurate than the extrapolation for the unified cache. Overall, these graphs indicate that the dilation model is quite accurate.

5.4 Actual versus dilated miss rates

Figure 5 presents the bottom line comparison between the actual misses, the dilated misses and the estimated misses. Each set of three bars corresponds to a particular processor as indicated on the X-axis. The misses are normalized with respect to the actual misses on the 1111 reference processor. The actual misses bar indicates the normalized misses obtained by simulating the caches on the actual traces generated by a particular processor. The dilated misses bar indicates the normalized misses obtained by simulating the caches on the reference trace, where each block is dilated by the text dilation. The estimated misses bar indicates the normalized misses obtained using our dilation model, assuming the text dilation factor. The difference between the first and last bars indicates the total error due to using our approach as opposed to actual simulation of each trace. The difference between the first and second bars indicated the error introduced by our assumption that all blocks are uniformly dilated by the text dilation. The difference between the second and third bars is due to the error introduced by our estimation of the misses on the dilated trace through our dilation model as opposed to simulation of the dilated trace.

In general, the instruction cache estimated misses track the actual misses very closely. Both the dilated and estimated misses are slightly more than the actual instruction cache misses. On the other hand, the unified cache misses do not track as well. The dilated misses are more than the actual misses, whereas the estimated misses are less than the actual misses. As we observed earlier, the model for the unified caches is less accurate, thus producing the larger estimation errors.

Given that it is infeasible to simulate all possible combinations of processors and caches for interesting design spaces, the alternative to using our dilation model is to assume that the memory hierarchy performance does not change as the issue width of the processor is increased. This assumption is equivalent to assuming that the normalized misses remains at one regardless of issue width. As the above figure shows, the actual normalized misses is much more than one, reaching up to six in certain cases for 085.gcc. This figure illustrates that one must account for the changes in memory hierarchy performance with issue width. This figure also illustrates how our dilation model can to a large extent account for these changes.

Table 3 presents similar results for all the benchmarks we have evaluated. There are four tables, one for each of the four cache configurations. Each row in the table corresponds to a particular benchmark. The actual, dilated and estimated misses are grouped into columns, where each group represents a particular processor design. The misses are normalized to unity for the reference processor. There is a large variation in the accuracy of the estimation across benchmarks and processor designs. The estimates track the actual misses better for narrower processors than for wider processors and better for instruction caches than for unified caches. There are some cases where the actual, dilated and estimated misses for the 6332 processor are far apart.

6 Conclusions

This work is motivated by the desire to automatically explore a large design space consisting of a cross-product of the processor design space and the memory hierarchy design space. We cannot exhaustively explore such a large design space in a timely manner. Therefore, we are forced to a hierarchical approach of partitioning the system and evaluating each component individually. Changes in processor architecture affect memory hierarchy performance but the constraints on evaluation time do not allow simulation of the traces for each processor architecture with each memory hierarchy design.

Instead, we adopt the approach of only simulating the caches on the traces from a reference processor. But, we model the traces of other designs as a dilated version of the reference processor's trace, where each block of instruction addresses is stretched out by the dilation coefficient. We use the ratio of code sizes with respect to the reference processor as the dilation coefficient. We use the AHH model to estimate the effects on instruction and unified cache behavior due to dilation of the reference trace. We evaluate the effectiveness of the dilation model and quantify the error due to each of the steps. The results show the model is highly effective for most of the processor design space.

7 Acknowledgements

The authors wish to thank all the members of the Compiler and Architecture Research (CAR) group at Hewlett Packard Laboratories who were engaged in the PICO automated embedded system design project. In particular, we thank Bob Rau for discussions that led to the dilation model and Greg Snider for architecting the software infrastructure of the spacewalker. We thank John Gyllenhaal at the University of Illinois for his help with the Impact emulation tools.

8 References

[1] J. Hoogerbrugge and H. Corporaal, "Automatic synthesis of transport triggered processors," presented at Proc. First Ann. Conf. Advanced School for Computing and Imaging, Heijen, The Netherlands, 1995.

[2] J. M. Mulder and R. J. Portier, "Cost-effective design of application-specific VLIW processors using the SCARCE framework," presented at Proc. 22nd Workshop on Microprogramming and Microarchitectures, 1989.

[3] D. Kirovski, C. Lee, M. Potkonjak, and W. M. Mangione-Smith, "Application-driven synthesis of core-based systems," presented at Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD), 1997.

[4] M. Kobayashi and M. H. Macdougall, "The Stack Growth Function: Cache Line Reference Models," *IEEE Transactions on Computers*, vol. 38, pp. 798--805, 1989.

[5] 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*, vol. 41, pp. 811-25, 1992.

[6] G. Rajaram and V. Rajaraman, "A probabilistic method for calculating hit ratios in direct mapped caches," *Journal of Network and Computer Applications*, vol. 19, pp. 309-319, 1996.

[7] D. B. Whalley, "Fast Instruction Cache Performance Evaluation Using Compile-Time Analysis," presented at Proc. ACM SIGMETRICS Conf., 1992.

[8] R. W. Quong, "Expected I-cache miss rates via the gap model," presented at Proc. 21 Int. Symp. Comp. Arch., 1994.

[9] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An Analytical Model for Designing Memory Hierarchies," *IEEE Transactions on Computers*, vol. 45, pp. 1180-94, 1996.

[10] K. Lee and M. Dubois, "Empirical models of miss rates," *Parallel Computing*, vol. 24, pp. 205-219, 1998.

[11] A. Agarwal, M. Horowitz, and J. Hennessy, "An Analytical Cache Model," *ACM Trans. on Computer Systems*, vol. 7, pp. 184--215, 1989.

[12] P. Steenkiste, "The Impact of Code Density on Instruction Cache Performance," presented at Proc. of 16th Intl. Symp. on Computer Architecture, 1989.

[13] S. Aditya and B. R. Rau, "Automatic architectural synthesis of VLIW and EPIC processors," Hewlett-Packard Laboratories, Palo Alto, CA to appear 1999.

[14] "Trimaran Compiler Infrastructure," `http://www.trimaran.org`.

[15] S. Aditya, B. R. Rau, and R. Johnson, "Automatic design of VLIW/EPIC instruction formats," Hewlett-Packard Laboratories, Palo Alto, CA to appear 1999.

[16] W. W. Hwu *et al*., "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229-248, 1993.

[17] R. A. Sugumar and S. G. Abraham, "Multi-configuration simulation algorithms for the evaluation of computer architecture designs," CSE Division, University of Michigan, Ann Arbor CSE-TR-173-93, 1993.

[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communication systems," presented at Proc. 30th Int. Symp. Microarchitecture, 1997.

1 KB Icache	1111	2111			3221			4221			6332		
Benchmark	Act	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est
085.gcc	1.00	1.22	1.46	1.45	1.75	2.12	2.08	2.02	2.43	2.40	2.89	3.50	3.45
099.go	1.00	1.53	1.46	1.41	2.13	1.96	1.92	2.55	2.30	2.23	3.66	3.31	3.23
147.vortex	1.00	1.30	1.46	1.42	1.84	1.88	1.85	2.14	2.15	2.12	3.01	2.95	8.45
epic	1.00	1.39	1.57	1.67	2.44	2.71	2.83	2.86	3.23	3.26	6.11	6.33	6.50
ghostscript	1.00	1.19	1.46	1.44	1.56	1.97	1.94	1.80	2.26	3.56	2.45	3.19	11.85
mipmap	1.00	1.20	1.47	1.37	1.55	2.03	1.97	1.79	2.87	2.88	2.35	3.20	3.22
pgpdecode	1.00	1.11	1.54	1.52	1.40	2.28	2.25	1.67	2.68	2.67	2.57	3.99	4.06
pgpencode	1.00	1.01	1.49	1.47	1.43	2.28	2.25	1.68	2.64	2.62	2.58	3.98	4.01
rasta	1.00	1.13	1.34	1.34	1.36	1.90	1.87	1.60	2.17	2.15	2.27	3.12	3.12
unepic	1.00	1.26	1.49	1.39	1.78	2.03	1.92	1.92	2.36	2.19	3.05	3.32	3.18

16 KB Icache	1111	2111			3221			4221			6332		
Benchmark	Act	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est
085.gcc	1.00	1.36	1.70	1.53	2.22	2.71	2.66	2.69	3.24	3.04	4.12	4.97	4.67
099.go	1.00	1.64	1.52	1.45	2.37	2.25	2.17	2.90	2.70	2.58	4.45	4.17	3.96
147.vortex	1.00	1.57	1.79	1.71	2.47	2.71	2.74	3.07	3.58	3.56	6.16	6.52	14.00
epic	1.00	1.34	1.54	1.77	2.37	3.65	3.58	3.12	4.30	4.38	5.75	9.37	17.35
ghostscript	1.00	1.39	1.89	1.77	2.28	2.99	3.06	2.79	3.83	5.71	4.20	7.38	21.80
mipmap	1.00	1.10	1.38	1.39	1.30	1.84	2.34	1.56	3.94	5.12	4.92	10.55	6.73
pgpdecode	1.00	1.72	2.31	2.10	2.68	4.77	4.60	3.48	6.32	5.89	6.22	11.73	11.76
pgpencode	1.00	1.49	2.19	1.92	2.60	4.51	4.47	3.37	5.89	5.48	5.96	10.91	10.74
rasta	1.00	1.17	1.27	1.22	1.49	1.85	1.76	1.75	2.15	2.14	2.74	3.69	3.63
unepic	1.00	1.45	1.58	1.84	3.32	3.58	3.37	2.77	4.70	4.39	6.57	9.49	10.63

16 K Ucache	1111	2111			3221			4221			6332		
Benchmark	Act	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est
085.gcc	1.00	1.26	1.49	1.10	1.85	2.14	1.25	2.15	2.52	1.32	3.12	3.65	1.55
099.go	1.00	1.47	1.38	1.10	1.95	1.86	1.23	2.31	2.16	1.31	3.32	3.09	1.55
147.vortex	1.00	1.28	1.42	1.03	1.76	2.03	1.10	2.15	2.34	1.14	3.58	3.78	1.26
epic	1.00	1.13	3.90	1.09	1.66	1.41	1.22	1.47	1.56	1.26	2.01	3.03	1.51
ghostscript	1.00	1.21	1.52	1.10	1.80	2.25	1.24	2.07	2.74	1.32	2.98	4.75	1.55
mipmap	1.00	1.02	1.42	1.16	1.27	1.63	1.46	2.31	4.02	2.01	4.91	7.05	2.25
pgpdecode	1.00	1.39	1.90	1.36	2.01	3.43	1.98	2.47	4.45	2.30	4.03	7.49	3.50
pgpencode	1.00	1.24	1.73	1.35	1.95	3.13	2.07	2.37	4.04	2.42	3.87	6.92	3.77
rasta	1.00	1.12	1.23	1.17	1.39	1.68	1.43	1.59	1.94	1.56	2.36	3.16	2.02
unepic	1.00	1.08	1.10	1.06	1.57	1.41	1.15	1.49	1.59	1.19	2.04	2.40	1.35

128 K Ucache	1111	2111			3221			4221			6332		
Benchmark	Act	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est	Act	Dil	Est
085.gcc	1.00	1.30	1.67	1.39	2.39	3.30	2.14	3.16	3.96	2.60	6.06	7.22	4.61
099.go	1.00	1.98	1.90	1.38	3.28	3.35	1.96	4.28	4.07	2.34	7.29	7.07	3.76
147.vortex	1.00	1.38	1.66	1.23	2.11	2.53	1.51	2.76	3.12	1.72	4.81	5.94	2.48
epic	1.00	1.02	1.03	1.16	1.08	1.09	1.44	1.10	1.11	1.54	1.18	1.20	2.15
ghostscript	1.00	1.28	1.47	1.48	1.66	2.02	2.22	1.91	2.35	2.75	2.70	3.44	4.92
mipmap	1.00	1.01	1.23	1.52	1.20	1.29	2.53	1.19	2.21	5.35	1.15	2.51	7.00
pgpdecode	1.00	1.27	1.47	1.85	1.68	2.67	4.21	3.81	32.22	5.97	58.94	158.73	16.32
pgpencode	1.00	1.22	1.55	1.91	3.63	2.64	4.79	1.88	8.67	6.82	27.17	40.46	19.30
rasta	1.00	1.28	1.50	1.64	1.87	2.30	3.27	2.13	2.65	4.47	3.08	3.65	11.06
unepic	1.00	1.04	1.05	1.15	1.09	1.13	1.38	1.10	1.16	1.47	1.20	1.29	1.97

Table 3: Actual, dilated and estimated misses for all benchmarks