Journal of Circuits, Systems, and Computers, Vol. 11, No. 5 (2002) 1–17
© World Scientific Publishing Company

POWER ESTIMATOR DEVELOPMENT FOR EMBEDDED SYSTEM MEMORY TUNING

FRANK VAHID[*,†,‡], TONY GIVARGIS[†,§] and SUSAN COTTERELL[*,¶]

[*] *Department of Computer Science and Engineering,*
University of California, Riverside,

[†] *Center for Embedded Computer Systems,*
Department of Info. & Computer Science,
University of California, Irvine, CA 92697, USA
[‡] *vahid@cs.ucr.edu*
[§] *givargis@ics.uci.edu*
[¶] *susanc@cs.ucr.edu*

Memory accesses account for a large percentage of total power in microprocessor-based embedded systems. The increasing use of microprocessor cores and synthesis, rather than prefabricated microprocessor chips, creates the opportunity to tune a memory hierarchy to the one program that will execute in the embedded system. Such tuning requires fast and accurate estimation of the power and performance of different memory configurations. We describe a general three-step approach to developing such estimators, based on our experiences on several different projects. Each step is increasingly fast, using the previous step to gauge accuracy. The first step uses high-level functional simulation, the second step uses trace simulation, and the third step uses equations. A tool developer can follow these three steps to create a powerful environment for core users to support synthesis of the best memory hierarchy for a particular embedded system. The approach can be applied to components other than memory also.

Keywords: System on a chip; platforms; memory; cores; low power; low energy; tuning; customized processors; configurable architecture.

1. Introduction

Accesses to instruction and data memory in a microprocessor-based system can consume a significant amount of total system power, nearly 50% for several common processors.[1,2] Thus, increased attention has been placed on reducing memory-related power. Various efforts have focused on designing low-power cache architectures,[3–9] on introducing tiny filter[10] or loop,[11–14] caches to reduce accesses to the regular memory hierarchy while executing small loops, on encoding bus traffic to minimize dynamic bus power,[15–17] on compressing instructions[18–20] and data[21–23] to reduce storage requirements and bus traffic, on compiling to reduce memory accesses,[24] and more. Memory access is also a key contributor to overall system performance.[25]

Meanwhile, modern core-based design methods enable designers to tune an architecture to a given program. In a typical embedded system, such as a set-top box or a digital camera, the program running on a microprocessor is fixed, or at least the program's general characteristics are well known. Ideally, a designer would be able to tune a microprocessor system to best execute that fixed program, or at least to a set of typical programs that might run on the microprocessor. Core-based design methods enable such tuning. In core-based design, a designer integrates processor-level components, like a microprocessor, memory, and peripherals, in an HDL (hardware-description language) environment. Once satisfied with the design, the designer fabricates an integrated circuit (IC). Core-based design contrasts sharply with standard design practice in the past, in which designers purchased existing ICs. Those existing ICs were designed to perform best for a large set of programs, but not for any one program in particular.

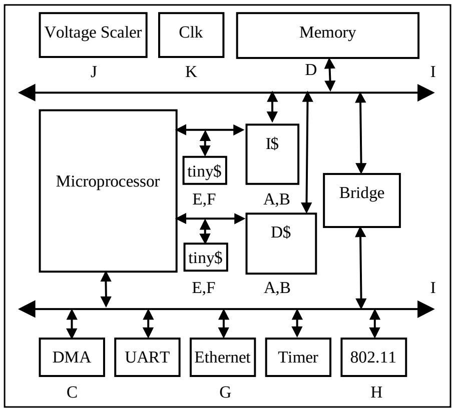
With the advent of core-based design, much recent research and commercial tools have focused on tuning the microprocessor instruction set to one fixed program.[26–31] In our work, we focus on the complementary problem of tuning the memory hierarchy to a fixed program.

Several related efforts demonstrate the benefits of tuning the memory hierarchy to a particular program. Dutt and Panda used an exploration strategy to find the best configuration of on-chip scratchpad memory size and certain cache parameters.[32,33] Kavvadias *et al.* created additional layers of small memories to store frequent data to reduce power.[34] Nachtergaele *et al.* presented an exploration environment that utilizes a two phase memory exploration scheme along with system level transformations to reduce memory size and power.[35] Shiue and Chakrabarti reduced power consumption by reducing memory traffic using memory optimizing transformations, storing frequently accessed variables in register files and on-chip cache, reducing misses by configuring the cache size correctly and by good data placement.[36]

In this paper, we define the problem of memory tuning and discuss the need for a memory tuning tool, we describe the three steps to developing a fast memory tuning tool, and we highlight results of various experiments.

2. Memory Tuning

We have investigated the problem of developing a memory tuning environment in the context of a parameterized platform. A *platform* is a pre-integrated design of processor-level components, components such as microprocessors, caches, memories, coprocessors, peripherals, and buses. We focus on the platforms that come in the form of intellectual property (IP), typically captured in an HDL, referred to as IP platforms. An IP platform may come in a synthesizable HDL form or a lower-level form, such as a gate-level HDL form, or even a layout form. A *parameterized platform* is a platform whose components come with configurable features that can be set to one of a limited number of values in order to set the component's operating mode, as shown in Fig. 1. For example, a cache may have several configurable features, including total size, line size, and associativity. A bus may have a configurable data encoder that can be activated or deactivated.



Parameter	Description	Configuration
A	Cache capacity	1 – 256K byte
B	Cache line size	32, 64, 128 byte
C	Data block size	32, 64 byte
D	Memory size	1-16M
E	Tiny cache size	8 – 256 entries
F	Tiny cache type	filter, loop, preloaded loop
G	Ethernet transfer rate	10M, 100M, 1G - bit
H	802.11 transfer rate	1M, 2M, 11M – bit
I	Bus encode/decode scheme	T0, bus invert, none
J	Supply voltage	1.5 – 2.5 Volts
K	Clock Frequency	50 – 300 MHz

Fig. 1. Parameterizable platform and the corresponding configurations.

A voltage source may be configurable to several voltage levels, while a clock may be configurable to different frequencies. A peripheral may have configurable buffer sizes, resolutions, or operating modes. A particular parameter setting for a component may result in a new customized HDL or layout representation being generated for that component. For example, a cache of a particular total size, line size, and associativity may be generated. In particular, the parameterization of the platform will not exist in the final version of the platform — instead, a particular customized instance of the platform will be generated.

IP platforms typically come with numerous configurable components. However, platform developers typically leave the platform user on his/her own to choose the best configuration of the platform's parameters. Instead, platform developers typically provide, in addition to basic software design tools, simulation support for the platform. The lowest-level design of platform, such as a gate-level design, can typically be simulated in an HDL environment. Likewise, the synthesizable version of the platform, if provided, can also be simulated in an HDL environment. Because such simulations are extremely slow, platform developers often provide even higher-level simulators, such as non-synthesizable high-level behavioral HDL models, or even functional simulators written in perhaps C or C_{++}. These higher-level simulators are functional only — while mirroring the lower-level representations, one cannot automatically derive an efficient lower-level implementation from these higher-level models.

In addition to simulation models, the platform developer may provide a menu-driven tool for selecting a particular configuration of parameterized components. This tool typically does not provide any guidance as to what the best configuration might be, but does eliminate the need for the platform user to modify HDL code. Such a tool may even generate customized software drivers for the particular platform configuration. Thus, the details of creating a customized platform instance are hidden.

However, the platform user must still determine the best configuration of the platform for a given program. The user is on his/her own in this respect — the user typically takes an educated guess as to the best configuration, or may run a few high-level simulations to compare some configurations he/she considers likely candidates. Unfortunately, finding the best configuration is a difficult task. There may be billions of possible configurations, with delicate relationships among the various parameters. For example, cache line size can have a tremendous impact on system performance depending on a particular program's behavior, and that line size can heavily impact bus traffic and hence has a relationship with any bus parameters. We use the term *tuning* to refer to the task of selecting the best configuration, in terms or power, performance, area and other metrics, of a platform's parameters considering the relationships of these parameters to a program's behavior and the relationships among parameters. A properly selected set of parameters can yield perhaps order of magnitude differences in terms of power and performance, while having a big impact on system area, compared to un-tuned parameters.[37]

Based on the above, we see a need for an automated tuning tool for parameterized platforms. Such a tool would take a given program, and find the best parameter configuration for that program's behavior and a particular set of design constraints.

Such a tool has two main parts — exploration methods, and estimation methods, as shown in Fig. 2. Exploration methods guide the search through the huge configuration space, narrowing the space down to the best set of candidate configurations. Exploration methods differ with respect to runtime and quality — longer running methods typically yield better quality. Ideally, the exploration tool will output a set of Pareto-optimal configurations — configurations such that no other configuration is better in all design metrics. That set represents the set of configurations with meaningful tradeoffs among the metrics.

Exploration requires methods of evaluating candidate configurations. Those methods are estimation methods. The estimation methods return information on power, performance, size, and other design metrics, for a given program executing on a given configuration. As with exploration, estimation methods differ with respect to runtime and quality — longer running estimation methods typically yield better accuracy.

However, in the case of both types of methods, quality does not only come from longer runtimes, meaning more complex algorithms. Instead, careful design of a method can also yield better quality. Thus, careful design of an exploration method that incorporates problem knowledge into the algorithm can often yield excellent results in short runtimes (e.g., carefully designed algorithms for solving the complex and well-known traveling salesman problem can solve very large problem sizes quickly).

Just as effort can be placed on developing problem-specific exploration methods to obtain quality results in reasonable runtimes, effort can be placed on developing estimation methods. A platform developer can focus on creating increasingly fast

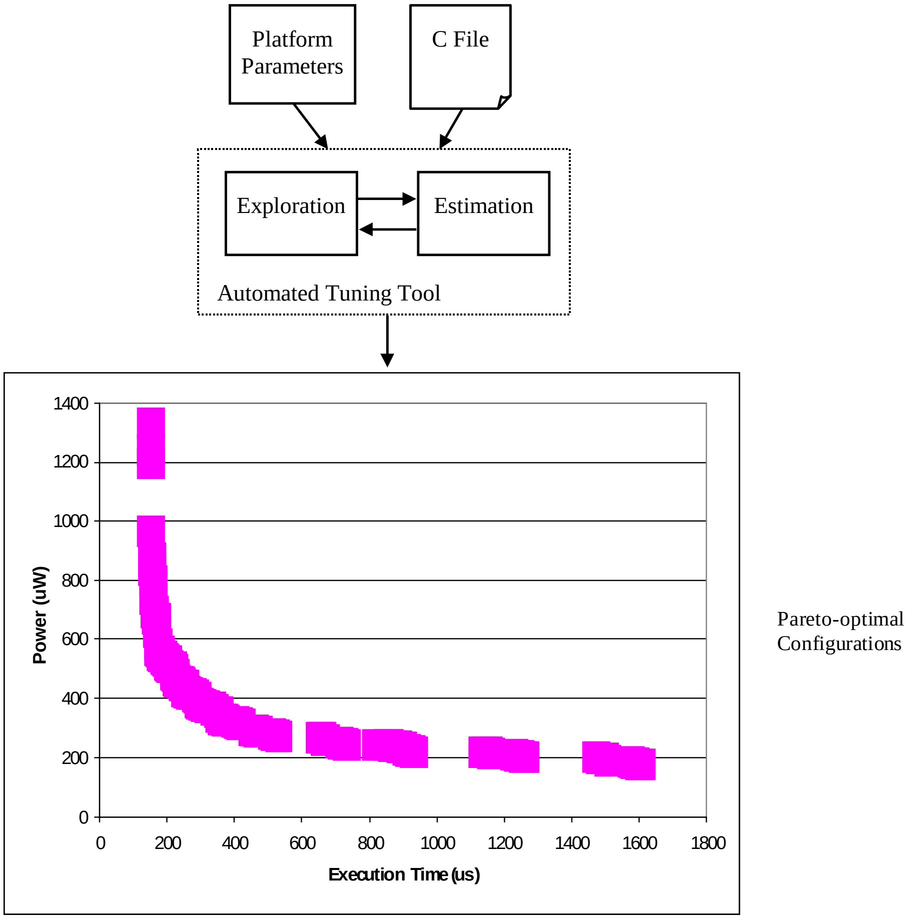


Fig. 2. Design methodology.

but still high-quality means for quickly determining the power, performance, and size of platform configurations.

In our work of developing parameterized memory components in the context of platforms, we have developed a general three-step approach that platform developers can follow to build increasingly fast estimators for their platforms.

We have looked at two types of parameterized memories. One type is a parameterized regular (level 1) cache architecture, with the cache parameters including total size, line size, and associativity. The other type includes parameterized filter and loop cache architectures, with the parameters including selecting between filter and loop cache styles, cache sizing, and selecting the number of supported loops. A filter cache[10] is an extremely small level 0 direct-mapped cache (e.g., 32 to perhaps 512 instructions) that will have a high miss rate, but an extremely low power per hit that in turn results in reduced overall energy for program execution. A loop cache[13] is also a small level 0 cache, but is only filled when a simple loop is detected in the instruction stream. By using a simple controller that detects loop entries and exits, tag comparisons can be completely eliminated in a loop cache, and misses are also completely eliminated.

Our three step approach consists of high-level functional simulation, trace-based simulation, and equation-based estimation, providing increasingly fast methods for estimating power and performance. The approach is summarized in Fig. 3. We now describe each step, and describe how we applied each step to our two types of parameterized memories.

3. High-Level Functional Simulation

A key idea of tuning is that the best parameter configuration for a platform depends not only on static constraints on the design metrics of power, performance, size, etc., but also on the dynamic behavior of the particular program mapped to the platform.[38] Thus, to determine the design metric values for a particular configuration, some form of simulation will be necessary. Though a platform typically comes with a gate-level or register-transfer level HDL representation, performing gate-level or even register-transfer level simulation for each configuration is very slow. Simulating even just one second of real time may take tens of hours or even days for any reasonable-sized platform. (Size can usually be determined from the configuration alone without simulation, but power and performance require simulation).

Thus, a platform developer should provide (and typically already does provide) a high-level simulation tool for a platform, as illustrated in Fig. 3. Though behavioral-level HDL code is faster than register-transfer or gate level, even faster are C/C_{++}/Java simulators. Such simulators may execute 1 second of real time in just tens of minutes. Those simulators are typically created to verify functionality and to provide performance data. They typically consist of a program module for each platform component. For example, Fig. 4 shows a simplified high-level

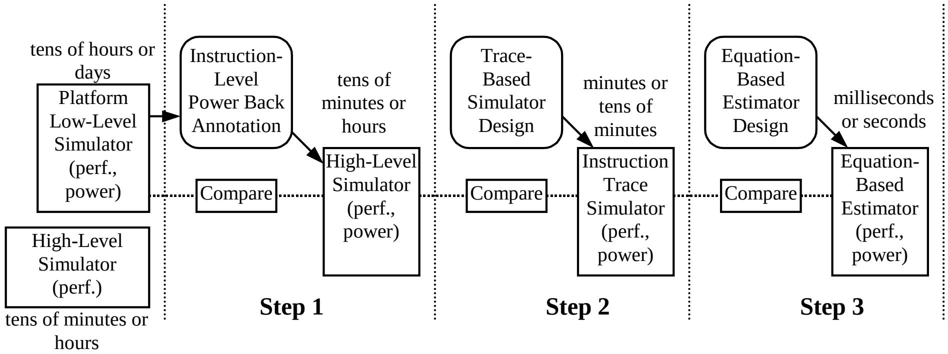
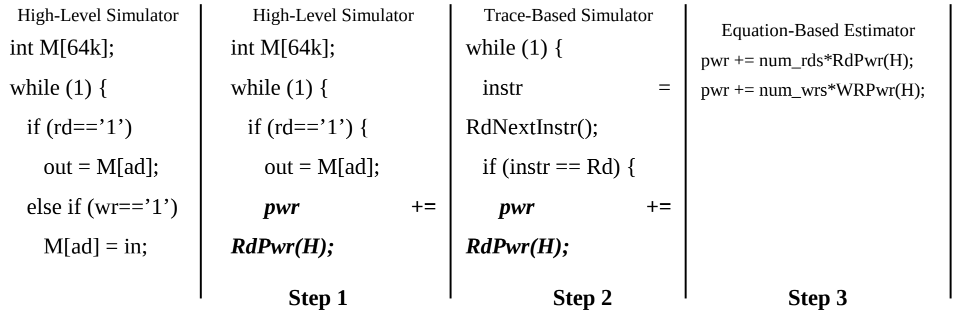

tens of hours or days
Platform Low-Level Simulator (perf., power)
High-Level Simulator (perf.)
tens of minutes or hours
Instruction-Level Power Back Annotation
tens of minutes or hours
Compare
High-Level Simulator (perf., power)
Step 1
Trace-Based Simulator Design
minutes or tens of minutes
Compare
Instruction Trace Simulator (perf., power)
Step 2
Equation-Based Estimator Design
milliseconds or seconds
Compare
Equation-Based Estimator (perf., power)
Step 3

Fig. 3. Three step approach for developing fast tuning methods: Step 1 — High-level functional simulation; Step 2 — Trace-based simulation; Step 3 — Equation-based estimation.



High-Level Simulator

```
int M[64k];
while (1) {
  if (rd=='1')
    out = M[ad];
  else if (wr=='1')
    M[ad] = in;
```

High-Level Simulator

```
int M[64k];
while (1) {
  if (rd=='1') {
    out = M[ad];
    pwr            +=
RdPwr(H);
```

Trace-Based Simulator

```
while (1) {
  instr            =
RdNextInstr();
  if (instr == Rd) {
    pwr            +=
RdPwr(H);
```

Equation-Based Estimator

```
pwr += num_rds*RdPwr(H);
pwr += num_wrs*WRPwr(H);
```

Fig. 4. Power estimator example for a memory M with one simple parameter H that selects between high performance/power mode and low power/performance mode.

simulator for a basic memory. The simulator declares a variable representing the memory, and then based on input read and write signals, the simulator either reads or writes the memory variable. Thus, the simulator preserves the functionality of the memory. We refer to such simulators as *functional* simulators.

In Step 1 of our approach, the platform developer extends such a high-level simulator to also evaluate power, as shown in Fig. 4, using back-annotation. The developer first determines the basic operations of the component for which power must be measured.

For a memory, those operations may include reads and writes. For a cache memory, those operations may be broken down further into read hits and read misses, and write hits and write misses. The developer must then determine the power for each such operation, for each possible parameter configuration. Such power determination may be done through an understanding of layout issues, through multiple simulations for different configurations, or through a combination of these two approaches.

In our efforts for regular caches, we deduce a physical model based on the cache parameter settings and technology feature size, similar to the approach used in CACTI models.[39] The physical model allows estimation of bit-line, word-line, comparator, storage transistors, and address decoding logic capacitive loads. Then, switching activity from the simulation phase is applied to obtain average power consumption of the cache for its various operations. We then annotated a high-level cache simulator with this power data.

We also applied the back annotation approach for our filter and loop caches. A filter cache is essentially a very small level 0 direct-mapped cache, and thus we simply used the same approach as for regular cache. However, loop caches are quite different from regular caches. Loop caches come in several varieties.[12] A *dynamic* loop cache[13] detects a short backwards branch in the instruction stream; such branches usually represent the end of a small loop. Hence, the branch triggers the filling of the loop cache during the second iteration of that loop (note that no processor stall occurs during this fill — instructions are simply copied from the instruction bus during execution). On the third iteration, instruction fetching switches from the power-costly instruction memory, which may be cache or a regular memory, to the very small low-power loop cache. Fetching continues from the loop cache until a control of flow change within the loop is executed. Another variety of loop cache, known as a *preloaded* loop cache,[12] gets preloaded with the most frequent loops as determined through profiling. Such preloading has the advantage of supporting control of flow changes within the loop (dynamically-loaded loop caches only fill what they saw on the second loop iteration, so cannot handle flow changes), thus supporting a wider range of loops and hence reducing power further. A *hybrid* loop cache[40] combines dynamic and preloaded loop caching, by only preloading those loops that do execute control of flow changes, and dynamically loading the rest, thus increasing the effective size of the preloaded loop storage.

We developed a functional loop cache simulator able to simulate any of the above loop cache varieties. Additional configuration information that the simulator could take included the size of the loop cache, the number of loops supported (for a preloaded or hybrid type), and miscellaneous options for each loop cache type.

We then proceeded to back-annotate the loop cache simulator with power information, by also deducing a physical model for the storage, as done for regular cache above. Furthermore, we had to determine the power for the loop cache controller. To do this, we first synthesized a variety of controllers and examined the power consumed by their various parts. We then determined the dependence of that power on the various configurations of the loop cache, including number of loops supported, fill strategy, etc.

The platform developer extends a high-level functional simulator by adding in calls to power estimation routines, as shown in Step 1 of Fig. 4. Each determined operation of the component will have its own routine. Each routine will have the current parameter configuration passed to it. The routine will then return a power value, and the simulator simply accumulates these power values as it executes.

The high-level simulator can now compute power and performance as it executes a program. The simulator can be incorporated with a configuration selector as shown in Fig. 5, which selects candidate configurations to evaluate. Such selection may be done manually by the platform user, or using automated search heuristics. However, such heuristics are limited in their search by the slowness of evaluation — executing a program using a functional simulator for a given configuration may take tens of minutes or even hours. Thus, those heuristics can only try tens of possible configurations.

We can also apply our approach to other components in a platform, such as processors, peripherals and buses. For a processor, an instruction based power modeling is applied that is based on models developed in Refs. 41 and 42. Similarly, for each bus segment, a rough layout is inferred that is based on the chip technology, chip area, bus widths, and relative size of the various cores, in order to obtain the average bus capacitance. Then, switching activity from the simulation phase is applied

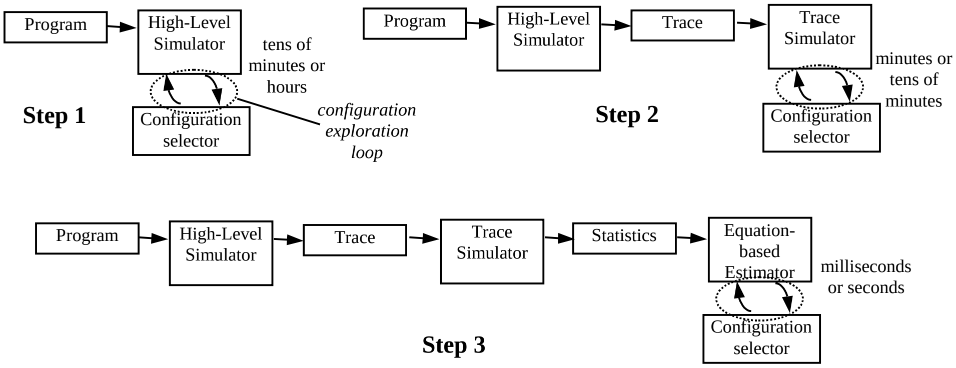


Fig. 5. Evaluating configurations.

to obtain average power consumption of various buses. Average accuracy of a high-level simulation based technique was experimentally shown to be 5% to 15% of gate-level measurements.[37] We apply a similar method for peripherals.[43]

4. Trace-Based Simulation

Although high-level functional simulations are far faster than lower-level simulations, the tens of minutes or hours required per simulation limits exploration methods to examining only a few configurations. Thus, we sought to develop a method that would provide reasonable accuracy in less time.

Most of the execution time of a high-level simulator is spent emulating the functionality of the platform. For example, in Step 1 of Fig. 4, reading and writing of the memory variable takes time. Simulating more complex functionality, such as cache fills, or loop cache control, takes even more time. However, notice that the simulation of that functionality is not really necessary for determining the power or performance. For those metrics, we really just need to know how many times each operation is carried out.

Developers of cache simulators have long recognized this principle. Hence, they developed trace-based cache simulators.[44,45] In such an approach, a functional simulator generates a trace of memory address references as the simulator executes. Once this trace is generated, the trace-based cache simulator can be executed multiple times with different configurations of common cache parameters, such as line size, associativity, total size, replacement policy, write policy, etc. The trace-based cache simulator does not maintain the actual data stored in the cache. Instead, it merely maintains the tags of items in the cache, and thus can determine whether an access would represent a hit or a miss. Not only is such trace-based simulation faster than a functional cache simulation, but trace-based simulation does not require re-simulation of the rest of the system for different cache configurations. We therefore developed a trace-based cache simulator that could support all the parameters we needed for our platform.

For example, Step 2 of Fig. 4 shows how the earlier functional memory simulator would be modified to become a trace simulator; notice that the functional aspects of the simulator have been removed, while the power estimation aspects remain. Thus, we can obtain power and performance data for each cache configuration in minutes or tens of minutes, as illustrated in Step 2 of Fig. 5. Notice that the time-consuming functional simulation is only done once, and is not in the main configuration exploration loop as is the case for Step 1, shown in Fig. 5.

We also developed a trace-based simulator for our loop cache. In this case, we modified the functional simulator to generate a trace of the instruction opcodes and addresses, rather than just the addresses as for the regular cache simulator. The trace-based loop cache simulator processes each instruction and determines for that instruction whether the loop cache will be idle, or will perform a detect operation, a fill operation, or a fetch operation. Using the back-annotated information, the trace-based loop cache simulator computes power.

Further methods can be applied to speed up such trace simulators, such as trace compaction,[46] trace stripping,[47] or evaluating multiple configurations in a single trace simulation.[45] For our loop cache, a simple method of reducing trace size was to only include branch instructions in the trace — the loop cache simulator could determine how many instructions existed between branches simply through address calculation.

Despite methods to reduce trace file size, one of the main disadvantages of a trace-based approach is that the trace files can become extremely large — many gigabytes in the Mediabench benchmarks we tried.

We have also developed trace-based simulators for the bus and processing components of a platform.[37,43] However, care must be taken to regenerate trace files when a configuration change demands such regeneration. For example, changing a cache's parameters will change the bus traffic between cache and memory, requiring a new bus traffic trace to be generated. Likewise, changing the resolution of a JPEG encoder will change the memory access patterns. A platform developer must carefully consider the impact of different configurations on the system's execution, and may have to regenerate new traces for certain classes of configurations. Our Platune environment allows a designer to capture the interdependency among parameter information as a directed graph,[48] and then automatically generates new traces when necessary during exploration. This does bring high-level simulation into the configuration exploration loop, but thus far in our experiments, the number of such occurrences has been manageable.

5. Equation-Based Estimation

Trace-based simulation can reduce estimation time to just minutes, enabling exploration tools to examine perhaps hundreds of configurations. However, we would really like to explore thousands or tens of thousands of configurations to find the best configuration. In order to reduce estimation time further, we sought a method for eliminating all or most of the time-consuming simulations from the exploration loop. For this purpose, we developed equation-based estimators.

The basic idea of equation-based estimation is to statistically characterize the trace, such that we can combine those statistics with a particular configuration's values in an equation or function to compute power. For example, Step 3 of Fig. 4 shows an equation-based estimator that makes use of statistics on the number of reads and writes in the trace.

Such equation-based estimation is extremely fast, but may lose accuracy, since in many cases the statistical characterization loses information necessary for accurate prediction. Notice in Step 3 of Fig. 5 that functional simulation is executed once to generate a trace, and trace-based simulation is executed once to generate statistics. Neither of those simulations are in the configuration exploration loop.

For our regular cache, we determined that we actually needed to run the trace-based simulator six times, not just once, to generate statistics for six key cache configurations. From those six, we could interpolate remaining configurations with reasonably accuracy. We define the equation-based cache estimation problem as follows. Given a trace of memory references, we are to compute the number of cache misses,[a] denoted N, for all different caches. Two caches are different if they differ in their total cache size S, line size (block size) L or degree of associativity A. We limit each of these three distinguishing parameters to a finite range:

$$\begin{aligned} &S = \{2^i, i = S_{\min} \cdots S_{\max}\}\,, \quad L = \{2^i, i = L_{\min} \cdots L_{\max}\}\,, \\ &A = \{2^i, i = A_{\min} \cdots A_{\max}\}\,. \end{aligned}$$

Note that, for practical purposes, we consider only values that are powers of two for each of these parameters. Given a trace-file, we must define a function:

$$f : S \times L \times A \to N\,.$$

To compute the number of cache misses N for any cache configuration. We assume that, with the aid of a cache simulator, we are able to compute the above function, for any value from the sets S, L and A, in linear time with respect to the size of the trace-file. Intuitively, our approach works as follows. We know that at low cache sizes, higher line size and associativity have a greater positive effect than they do at high cache sizes. For example, doubling the line size when cache size is 512B may reduce cache miss rate by 30%, but when the cache size is 8 K, it may not reduce the miss rate at all. Thus, we are interested in finding these improvement ratios at both low and high cache sizes, so that, by line fitting, the improvement ratio for any cache size can be estimated. This assumes a smooth design space between these points. We next describe our approach for estimating this function for all range values.

[a] Other metrics, e.g., number of write backs, can be estimated, using our approach, in a similar manner.

Our approach consists of three steps. First we simulate the trace-file for some selected S, L and A values and obtain the corresponding cache misses. Then we calculate a linear equation, using the least square approximation method. Last we use our linear equations to compute N for all cache parameters. We first simulate the following points in our domain space:

$$f(S_{\min} \times L_{\min} \times A_{\min}) = N_1\,, \quad f(S_{\max} \times L_{\min} \times A_{\min}) = N_2\,,$$
$$f(S_{\min} \times L_{\max} \times A_{\min}) = N_3\,, \quad f(S_{\min} \times L_{\min} \times A_{\max}) = N_4\,,$$
$$f(S_{\max} \times L_{\max} \times A_{\min}) = N_5 \quad f(S_{\max} \times L_{\min} \times A_{\max}) = N_6\,.$$

Then we compute the following ratios:

$$R_1 = N_1/N_3\,, \quad R_2 = N_1/N_4\,, \quad R_3 = N_2/N_5\,, \quad R_4 = N_2/N_6$$

Here, R_1/R_2 denotes the improvement we obtain by using maximum line-size/associativity when cache size is at its minimum. Likewise R_3/R_4 denote the positive improvement we obtain by using maximum line-size/associativity when the cache size is at its maximum. Given these ratios we estimate N for a given cache size S, line size L, and associativity A as follows:

$$s = (S_i - S_{\mathrm{Min}})/S_{\mathrm{Max}}\,, \quad l = (L_j - L_{\mathrm{Min}})/L_{\mathrm{Max}}\,, \quad a = (A_k - A_{\mathrm{Min}})/A_{\mathrm{Max}}\,,$$
$$t_1 = s(N_2 - N_1) + N_1\,, \quad t_2 = l(R_3 - R_1) + R_1\,, \quad t_3 = a(R_4 - R_2) + R_2\,,$$
$$f(S_i, L_j, A_k) \approx t_1(1 - t_2 - t_3)\,.$$

The first three equations, s, l and a, normalize our parameters to be within a unit range. The next equation, t_1, estimates cache misses using lowest line size and associativity, by computing a linear line through the points N_1 and N_2. If more simulation data is available, the least square approximation is used to compute t_1. The next two equations, t_2 and t_3, estimate the expected improvement gained from higher line size or associativity. The last equation combines the previous equations to estimate cache miss rate.

Further details of our equation-based cache estimation can be found in Ref. 49.

We can apply a similar method for filter caches. However, loop caches require a very different approach. In our approach, we developed a tool to parse the trace file and generate a statistical characterization of the loop behavior of the program. For every loop, we compute statistics (average, minimum, maximum, and standard deviation) of the number of visits to this loop, the number of iterations of this loop per visit, and the number of instructions executed by this loop per iteration. The tool also examines the program code itself to determine the static size of each loop and the number of branch statements within the loop.

We then developed an estimation tool that tries to estimate the behavior of the various loop cache configurations based on the generated loop statistics. For example, suppose a loop's statistics indicate that the loop iterates 100 times per visit, with a standard deviation of 0. Suppose that loop executes 10 instructions per iteration, with a standard deviation of 0. We can see that this is likely a loop with a fixed iteration count and containing straight-line code. For a dynamically-loaded loop cache, we know that for each visit, this loop will generate 10 fill operations (during the second iteration), and then for the remaining 98 iterations, the loop will be fetched from loop cache, resulting in 98*10 = 980 fetch operations from the loop cache. For a preloaded loop cache, each visit will result in 100*10 = 1000 fetch operations.

We apply a similar process for all loop cache variations. We consider additional details, such as detect operations necessary for preloaded loop caches.

Note that the above approach can result in inaccuracy. For example, when the standard deviation of a loop's instructions per iteration is nonzero, we do not know how the iterations look across loop visits. We must make some assumptions.

To improve the accuracy, we can try to find additional statistics that would help — these are highly-dependent on the loop cache style, and thus this step requires careful attention by the platform developer.

6. Results

The three steps outlined above provide increasingly fast power estimation at the expense of some accuracy loss. We now highlight some data showing the speed and accuracy of the methods we developed for regular cache and for loop cache.

Figure 6 provides performance and energy (power times time) estimation data for our trace-based cache simulation approach compared with our equation-based estimation approach, for a regular cache executing a diesel engine controller example. That data also includes a configurable bus, for which trace and equation-based simulators were also developed.[50] We evaluated over 45 000 different configurations of the cache/bus system — the figure shows 10 of those configurations, selected to reflect worst, average and best case estimates. Notice that the equation-based method is quite accurate. For two different examples and all 45 000 configurations, average error was only 2%, and worst case error was 18%.[49,50] Obtaining these values for all possible cache/bus configurations using equation-based estimation required only

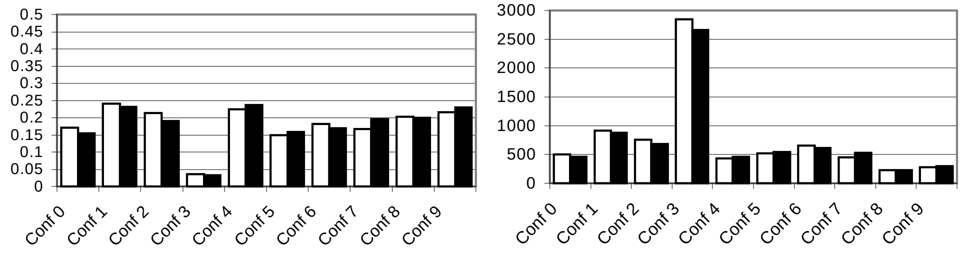


Fig. 6. Performance (left, in sec.) and energy (right, in mJ) estimates from trace-based simulation (white bars) versus equation-based estimation (black bars) for ten different regular cache configurations, using a diesel engine controller example running on a MIPS processor.

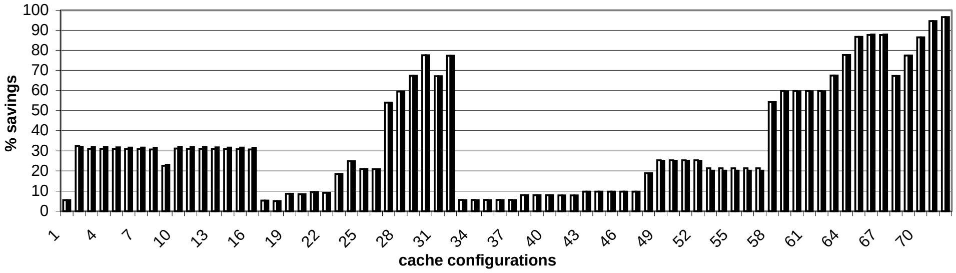


Fig. 7. Instruction-fetch power savings estimated by trace-based simulation (white bars) versus equation-based estimation (black bars) for 72 different loop cache configurations, executing the JPEG benchmark on a MIPS processor. Loop caching does not impact performance, so no performance estimates are shown.

84 minutes, instead of 7 days for the trace-based simulation approach — a speedup of 120 times.

Figure 7 summarizes power savings estimations for a JPEG decoder benchmark using a variety of loop cache configurations. We examined 72 different configurations, including different sizes and types of dynamically-loaded loop caches (configurations 1–16), of preloaded loop caches looking for a loop's starting address (configurations 17–32), and preloaded loop caches looking for a loop's ending address (configurations 33–72). The white bars represent the trace-based simulator results, while the black bars represent the equation-based estimator results, for each configuration. Notice that the equation-based method is extremely accurate — averaging only 1% error. We applied these methods to the PowerStone set of benchmarks,[2] and obtained an average error of only 2%. The trace-based loop cache simulator required an average of 300 seconds per configuration, while the equation-based estimator took less than 0.01 seconds — a speedup of 30 000.

In both of the above cases, we examined all configurations of the parameterized components. Related to the above work is work we have done to more efficiently search the configuration space, using knowledge of the parameter interdependencies to enable extensive search space pruning.[37,48]

Results thus far have focused on individual memory components and on certain combinations of processor, bus and memory. We plan in the future to create a comprehensive exploration tool based on the three step methodology, that simultaneously considers all of the parameters of Fig. 1.

7. Conclusions

A need exists for platform developers to provide tuning tools that assist platform users to select the best configuration of platform parameters. Platform developers can follow the three-step approach described in this paper to create fast yet accurate tuning tools. The first step involves creating high-level functional simulators (really, just extending existing such simulators) accumulate for each operation the power and performance data that has been back-annotated from low-level simulations. The second step involves modifying a high-level simulator to output instruction traces for every component, and developing trace simulators for each component. The third step involves developing equations that can predict the power and performance data from statistical summaries of the traces. With this third type of estimator, the platform developer can develop exploration methods that thoroughly search the configuration space, enabling the platform user to effectively tune the platform to a specific program. The net result is a lower power, higher performing, more size efficient synthesized platform implementation.

We are continuing to develop parameterized memory and bus components that provide good power/performance tradeoff capability for core-based systems. We are also investigating the idea of heavily parameterized pre-fabricated platforms, whose parameters would be configured by setting bits in registers on the chip. In particular, we are developing new highly parameterized memory components for such platforms, along with methods for tuning such components to a program.

Acknowledgment

This work was supported in part by the National Science Foundation (CCR-9811164), NEC C&C Research Labs, and a Dept. of Education GAANN fellowship.

References

1. J. Montanaro *et al.*, “A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor”, *IEEE J. Solid-State Circuits*, 1996.
2. J. Scott, L. Lee, J. Arends, and B. Moyer, “Designing the low-power MCORE architecture”, *Power Driven Microarchitecture Workshop ISCA*, 1998.
3. D. H. Albonesi, “Selective cache ways: On-demand cache resource allocation”, *J. Instruction level Parallelism*, 2000.
4. R. Bahar, G. Albera, and S. Manne, “Power and performance tradeoffs using various caching strategies”, *Int. Symp. Low Power Electronics and Design*, 1998.
5. C. Chakrabarti, “Cache design and exploration for low power embedded systems”, *Int. Performance, Computing, and Communication Conf.*, 2001.
6. Z. Hu, M. Martonosi, and S. Kaxiras, “Improving cache power efficiency with an asymmetric set-associative cache”, *Workshop Memory Performance Issues*, 2001.
7. K. Inoue, T. Ishihara, and K. Murakami, “Way-predicting set-associative cache for high performance and low energy consumption”, *Int. Symp. Low Power Electronics and Design*, 1999.
8. A. Malik, B. Moyer, and D. Cermak, “A low power unified cache architecture providing power and performance flexibility”, *Int. Symp. Low Power Electronics and Design*, June 2000.
9. C. Su and A. Despain, “Cache designs for energy efficiency”, *Proc. 28th Annual Hawaii Int. Conf. Syst. Sciences*, 1995.
10. J. Kin, M. Gupta, and W. Mangione-Smith, “The filter cache: An energy efficient memory structure”, *Int. Symp. Microarchitecture*, 1997.
11. N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, “Energy and performance improvements in microprocessor design using a loop cache”, *Int. Conf. Computer Design*, 1999.

12. A. Gordon-Ross, S. Cotterell, and F. Vahid, "Exploiting fixed programs in embedded systems: A loop cache example", *IEEE Computer Architecture Lett.*, 2002.
13. L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops", *Int. Symp. Low Power Electronics and Design*, 1999.
14. L. H. Lee, W. Moyer, and J. Arends, "Low-cost embedded program loop caching — revisited", *University of Michigan Techn. Report Number CSE-TR-411-99*, 1999.
15. L. Benini, G. Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address bus encoding techniques for system-level power optimization", *Design Automation and Test in Europe*, 1998.
16. J. Henkel and H. Lekatsas, "A2BC: Adaptive address bus coding for low power deep sub-micron designs", *Design Automation Conf.*, 2001.
17. M. Stan and W. Burleson, "Bus-invert coding for low-power I/O", *IEEE Trans. VLSI Syst.*, 1995.
18. L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective instruction compression for memory energy reduction in embedded systems", *Int. Symp. Low Power Electronics and Design*, 1999.
19. T. Ishihara and H. Yasuura, "A power reduction technique with object code merging for application specific embedded processors", *Design and Test in Europe*, 2000.
20. H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design", *Design Automation Conf.*, 2000.
21. G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Energy savings through compression in embedded java environments", *Int. Symp. Hardware/Software Codesign*, 2002.
22. J. Yang and R. Gupta, "FV encoding for low-power data I/O", *Int. Symp. Low Power Electronics and Design*, 2001.
23. J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches", *Int. Symp. Microarchitecture*, 2000.
24. M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power", *Design Automation Conf.*, 2000.
25. J. Hennessy and D. Patterson, "Computer architecture: A quantitative approach", 3rd ed., ISBN 1-55860-596-7, Morgan Kaufman, 2002.
26. S. Abraham, B. Rau, R. Schreiber, G. Snider and M. Schlansker, "Efficient design space exploration in PICO", *Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.
27. ARC International, "A platform approach to reducing time to market: The ARCform^{TM} SoC Development Platform and ARCtangent^{TM}-A4 customizable processor", http://www.arc.com.
28. Altera's NIOS processor, http://www.altera.com/products/devices/nios/nio-index.html.
29. J. Fisher, "Customized instruction-sets for embedded processors", *Design Automation Conf.*, 1999.
30. J. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit processors: Letting applications define architectures", *MICRO*, 1996.
31. R. Gonzalez, "Xtensa: A configurable and extensible processor", *MICRO*, 2000.
32. N. Dutt, "Memory organization and exploration for embedded systems-on-silicon", *Int. Conf. VLSI and CAD*, 1997.
33. P. Panda, N. Dutt, and A. Nicolau, "Architectural exploration and optimization of local memory in embedded systems", *Int. Symp. Syst. Synthesis (ISSS)*, 1997.

34. N. Kavvadias, A. Chatzigeorgiou, N. Zervas, and S. Nikolaidis, "Memory hierarchy exploration for low power architectures in embedded multimedia applications", *Int. Conf. Image Processing (ICIP)*, 2001.
35. L. Nachtergaele, F. Catthoor, F. Balasa, F. Franssen, E. DeGreef, H. Samsom, and H. De Man, "Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems", *Int. Workshop Memory Technol.*, 1995.
36. W. Shiue and C. Chakrabarti, "Memory design and exploration for low power, embedded systems", *J. VLSI Signal Processing — Syst. for Signal, Image, and Video Technol.* **29**, 3 (2001) 167–178.
37. T. D. Givargis, F. Vahid, and J. Henkel, "System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip", *Proc. Int. Conf. Computer-Aided Design*, November 2001.
38. T. Givargis, J. Henkel, and F. Vahid, "Interface and cache power exploration for core-based embedded systems", *Int. Conf. Computer-Aided Design (ICCAD)*, 1999.
39. G. Reinman and N. Jouppi, "CACTI2.0: An integrated cache timing and power model", *Compaq WRL Research Report 2000/7*, 2000.
40. A. Gordon-Ross and F. Vahid, "Dynamic loop caching meets preloaded loop caching — A hybrid approach", *Int. Conf. Computer Design*, 2002.
41. D. Brooks, V. Tiwari, Martonosi, and M. Wattch, "A framework for architectural-level power analysis and optimizations", *Proc. Annual Int. Symp. Computer Architecture*, 2000.
42. V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization", *IEEE Trans. VLSI Syst.* **2**, 4 (1994) 437–445.
43. T. Givargis, F. Vahid, and J. Henkel, "Trace-driven system-level power evaluation of system-on-a-chip peripheral cores", *Asia South-Pacific Design Automation Conf. (ASPDAC)*, 2001.
44. J. Elder and M. Hill, "Dinero IV trace-driven uniprocessor cache simulator", http://www.cs.wisc.edu/∼markhill/DineroIV/.
45. R. Sugumar and S. Abraham, "Efficient simulation of caches under optimal replacement with application to miss characterization", *Sigmetrics Conf. Measurement and Modeling of Computer Syst.*, 1993.
46. C. Tsui, R. Marculescu, D. Marculescu, and M. Pedram, "Improving the efficiency of power simulators by input vector compaction", *Design Automation Conf.*, 1996.
47. Z. Wu and W. Wolf, "Iterative cache simulation of embedded CPUs with trace stripping", *Int. Symp. Hardware/Software Codesign*, 1999.
48. T. Givargis and F. Vahid, "Platune: A tuning framework for system-on-a-chip platforms", *IEEE Trans. Computer Aided Design*, to appear 2002.
49. T. Givargis, F. Vahid, and J. Henkel, "Fast cache and bus power estimation for parameterized system-on-a-chip design", *Design Automation and Test in Europe (DATE)*, 2000.
50. T. Givargis, F. Vahid, and J. Henkel, "Evaluating power consumption of parameterized cache and bus architectures in system-on-a-chip designs", *IEEE Trans. VLSI* **9**, 4 (2001) 500–508.