Compiler Techniques for Energy Saving in Instruction Caches of Speculative Parallel Microarchitectures*

Seon Wook Kim Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
{seon,eigenman}@ecn.purdue.edu

Abstract

We present a new software scheme, called compiler-assisted I-cache prediction (CIP) *for energy reduction in instruction caches. With the help of compiler-supplied information, the processor is able to turn off substantial portions of the I-cache. The necessary cache sets are only turned on during the execution of individual code sections. The CIP scheme is based on the processor's ability to predict code sections that are about to execute and on the compiler's ability to precisely inform the hardware about the size of these code sections. Our techniques grew out of work with optimizing compilers for speculative parallel microarchitectures. The use of this target machine class is further motivated by the fact that speculative processors have the potential to overcome limitations in the compiler parallelization of many applications, especially non-numerical programs. Speculative microarchitectures are also among the most promising emerging architectures that can take advantage of the ever-increasing levels of chip integration. We will show that our new techniques can lead up to 90% I-cache energy savings in general-purpose applications without significant execution overhead. We believe that this is a substantial step towards the goal of making such chips integral parts of mobile computing devices, such as laptops, palm tops, and cellular phones.*

Keywords: *energy saving, speculative microarchitecture, compiler, instruction cache, branch prediction*

*This work was supported in part by NSF grants #9703180-CCR and #9872516-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

1 Introduction

In most modern high performance processors caches consume a major portion of the chip's power. Caches are typically implemented with SRAM cells and often occupy a large portion of the chip area. Power reduction in caches can be achieved through several techniques: semiconductor process improvements, memory cell redesign, voltage reduction, optimized cache structures, and *software assistance*. Our research focuses on software-assisted techniques for energy reduction in caches, which have great promise, but have not yet been explored extensively. Both instruction and data cache energy reduction need to be performed in order to reduce the power consumption of a processor chip to a very low level. In this paper we concentrate on the first one of these issues, I-cache energy reduction.

The proposed techniques are based on our previous work in parallel processing technology. Specifically, we have developed optimizing compiler techniques and, most recently, we have focused on the interface between compilers and speculative parallel microarchitectures [6]. We have found that these techniques lend themselves to power reduction methods, which we present in this paper.

We have chosen speculative parallel microarchitectures as the target machine class. This has several reasons. Speculative architectures have been proposed in order to overcome the limited success in automatically parallelizing non-numeric applications [9, 5, 10, 12, 13]. The ability to speculatively execute code sections in parallel relieves the compiler from the burden of providing absolute guarantees about program data dependences between sections of code believed to be parallel. When compiling for speculative microarchitectures, the compiler will explicitly identify parallel code sections. The remaining sections are considered as speculative regions because the compiler is not certain that these sections can be executed in parallel. It is then left to the speculative hardware to determine whether or not the parallelism among the speculative regions can be exploited.

Another motivation for our focus on speculative microarchitectures is that improvements in fabrication technology are resulting in ever-increasing levels of chip integration. Microarchitectures take advantage of these trends by distributing the execution resources into multiple processor cores, resembling conventional multiprocessors, and also provide hardware support for speculative execution [9, 5, 10]. The idea of speculative execution has been used in research and practice for several years. State-of-the-art processors primarily use control speculation in the form of branch prediction as an essential means to sustain today's processor speeds.

The specific architecture used in our work is the Wisconsin Multiscalar [9]. We will briefly introduce this architecture in Section 3. The main feature of this processor that our scheme exploits is its ability to predict executing tasks with high accuracy. We have extended the Multiscalar simulator to include small hardware extensions that complement our software energy-saving techniques. Specifically, they switch on and off individual sets of the instruction cache. While our techniques currently rely on specific features of the Multiscalar architecture, this is not a substantial restriction. We are developing extensions of our scheme to more general machine classes in ongoing work.

Our new power-saving scheme is called *compiler-assisted I-cache prediction (CIP)*. The compiler annotates each task in a program with its number of instructions. Tasks are the basic units of parallel speculative execution on the Multiscalar architecture. At runtime, the processor predicts with high accuracy the task that will execute next. This information plus the compiler-generated task size allows the processor to predict the I-cache sections that will be used next and turn on their power. When a section is no longer used, the power is again turned off. A distinguishing feature of this scheme is that we do not make any assumptions about the structure of the application program. Task prediction in our architecture works well in loop-based programs as well as in those with irregular control flow. This is one of the main aspects in which our work differs from related approaches, which focus on loop-based program structures [2, 1].

In Section 2 we discuss related research in energy saving technology; in Section 3, we describe our new compiler technique for energy saving. In Section 4 we analyze the performance, comparing with a state-of-the-art scheme in terms of the power reduction and the execution time overhead. Section 5 concludes the paper.

2. Related Work

The area of power minimization in caches at the architectural and software levels is relatively new. Several architectural approaches have been proposed and some are implemented in commercial microprocessors to reduce cache power dissipation by limiting switching activity. For example, instead of tag lookup in parallel with the access to the L2 cache, Alpha 21164 processors placed the tag lookup and data access in series. By looking up the tag first, the chip can use power only for the bank that contains the requested data, reducing cache power by roughly two-thirds [4]. However, this strategy gains major power reduction at the cost of two cycles in cache access.

There have been research efforts on techniques to reduce the power consumption in the memory hierarchy [2, 1, 3]. The impact of memory hierarchy in minimizing power consumption and the data-reuse for the power reduction in memory accesses are addressed in [3].

An small extra cache is added between the CPU and the L1 cache, called filter cache [7] to trade performance for power consumption. The filter cache delivers large energy gains at the expense of decreased hit ratio and, hence, longer average memory access time. Because filter caches incur a large execution time penalty, they are not suited for high performance processors.

In order to overcome the problems with filter caches, techniques were proposed in [2] that use an additional mini cache, called a loop cache (L-cache). It is located between the instruction cache and the CPU core, and it buffers instructions that are nested within loops and otherwise fetched from the instruction cache. The authors showed that the loop cache is much less and simpler than the instruction cache, and they proposed compiler techniques to allocate instructions into the L-cache. However, its benefits are limited in the case of large loops that do not fit in the L-cache.

The same authors in [1] use a branch prediction scheme for dynamically allocating instructions in an additional mini cache, called L0-cache, instead of relying on the compiler capability. They capture the most frequently executed basic blocks by inspecting branch frequencies. Only frequently executed code sections are then copied into the L0-cache. This scheme does not require code modifications by the compiler. One disadvantage is that non-loop regions and large loops cannot benefit from the L0-cache.

The performance of most power minimization techniques for memory hierarchies highly depend on the structure of the applications. These approaches have two major problems: One is that the allocated instructions fit in the mini cache. This is a problem for applications with large loops. Another one is that integer applications that have significant non-loop regions cannot benefit from this scheme.

In addition to the research in memory hierarchies, power minimization techniques have been proposed for speculative high-performance processors. The work in [8] focuses on the excessive energy dissipation of high performance speculative processors that execute more instructions than are actually needed in a program. The processor stops execution in the pipeline when there is a large probability of wrong path execution, and it resumes only when the actual execution path is detected.

In [11] several compiler techniques that are useful in power minimization are reviewed. It is shown that compiler optimizations, such as instruction reordering, code generation through pattern matching, and reduction of memory operands, are beneficial for the reduction of energy, because they reduce the running time of the code.

There is little research in generalized schemes to reduce the power dissipation in caches without excessive execution penalty. In the next section, we propose low-overhead compiler techniques for power reduction in an instruction cache on a specific architecture. We chose speculative microarchitectures for our work. Speculative processors are not only among the most interesting emerging architectures, they also already provide some of the capabilities that assist our power-saving compiler techniques. The extension of our scheme for conventional processors is straightforward.

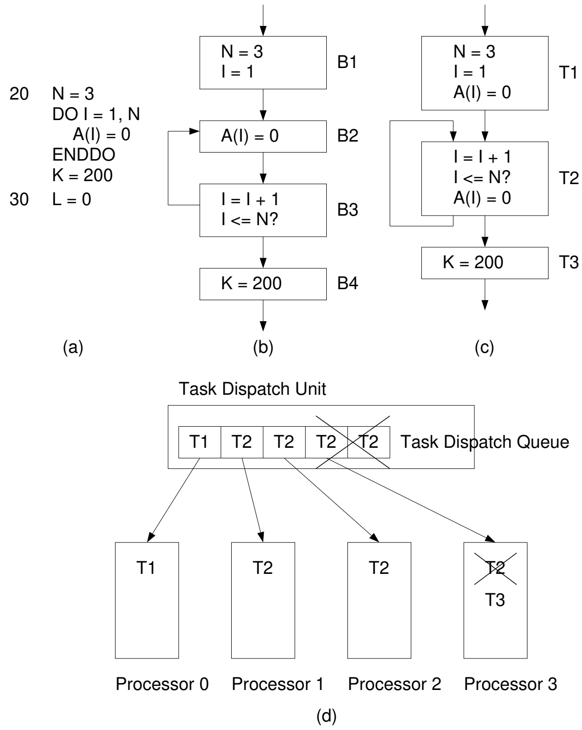

Figure 1. Multiscalar Execution. (a) Example Code. (b) Basic Blocks with CFG. (c) Task graph. (d) Execution on Multiscalar. `Bn` indicates a basic block, and `Tn` indicates a task. A single program is divided into a collection of tasks, and the processor walks through this structure task by task.

3 Compiler Techniques for Energy Saving in Instruction Caches

A major concern in implementing speculative microarchitectures is their high complexity and power consumption. We propose a new scheme, called *compiler-assisted I-cache prediction (CIP)*, to reduce power consumption in an instruction cache using compiler techniques and the built-in branch predictor on speculative processors, such as the Wisconsin Multiscalar architecture [9].

A basic understanding of the Multiscalar compilation scheme is useful to understand our energy saving mechanism. The Multiscalar code generator divides a single program into a collection of tasks, and the processor walks through the control flow graph (CFG) task by task as shown in Figure 1. A task is a portion of the CFG whose execution corresponds to a contiguous region of the dynamic instruction sequence. A program is statically partitioned into tasks, which are annotated to describe next tasks and register dependences. Figure 1 (b) shows the basic blocks (`B1` to `B4`) with control flows. The code generator walks through the CFG and generates tasks with one or more basic blocks. Figure 1 (c) shows that a task `T1` consists of two basic blocks, `B1` and `B2`, and a task `T2` consists of two basic blocks `B2` and `B3` respectively. And the code generator specifies the next tasks inside the task header: A task `T1` has one target `T2`, and a task `T2` has two targets, `T2` itself and `T3`. At runtime, the tasks are distributed to a number of processing units by a inter-task predictor which predicts the next tasks in the task dispatch unit, and the processor continues to walk from one point to the next point in the task graph. Figure 1 (d) shows an example of executing code on a Multiscalar architecture with four processors. The task dispatch unit fetches tasks using an inter-task prediction, and assigns tasks onto each processor. If the task is mispredicted, then the assigned task (a task `T2` on processor 3) is rolled back, and the new task is reassigned (a task `T3` on processor 3).

Table 1. Accuracy of Inter-task Prediction in SPEC95 Benchmarks. In most benchmarks, the accuracy of inter-task prediction is very high.

Benchmarks	Accuracy
SWIM	99.8%
HYDRO2D	99.8%
FPPPP	98.4%
COMPRESS	97.2%
GO	84.6%

3.1 Power Reduction through Compiler Techniques

As described above, on Multiscalar, a program is statically partitioned into tasks, and the tasks are distributed to a number of processing units. The inter-task predictor in hardware predicts the next task with high confidence. Table 1 shows the accuracy of inter-task prediction in several benchmarks. The prediction is very accurate except in the GO benchmark. The task dispatch unit has the responsibility to assign tasks to each processor.

Our new compiler technique expands a task header in order to include the task size. To this end the compiler analyzes the number of instructions in each task. At runtime, thanks to the knowledge of the task size and the (accurately predicted) next task, the processor knows exactly which sets of an instruction cache will be used in the near future. This is shown in Figure 2. Before a task is invoked, the task dispatch unit turns on the instruction cache from *the start address of a task (*An*)* to *the start address of a task (*An*) + size of the task (*In*)*. The start address is the predicted task address given by the inter-task predictor. When the task is completed or rolled back (because of incorrect inter-task prediction or the violation of data dependences) the cache sets for the task are turned off, unless the task is still needed. The task is still needed if it is currently executing on another processor or if it is found in the task dispatch unit, about to be re-executed. If the turned-off cache sets are accessed (i.e., the prediction was wrong), a *cache miss* results. The switching latency in this case will be hidden by the cache miss latency. The performance of our scheme depends on the task generation by the compiler and the accuracy of the hardware branch prediction. Because the compiler precisely knows the number of instructions in the task and the accuracy of the inter-task branch prediction is high, the performance is high as well. This is true even in non-loop regions with complex control flow. It is a distinguishing aspect of our scheme, and contrasts with other approaches that perform well only in loop-based programs.

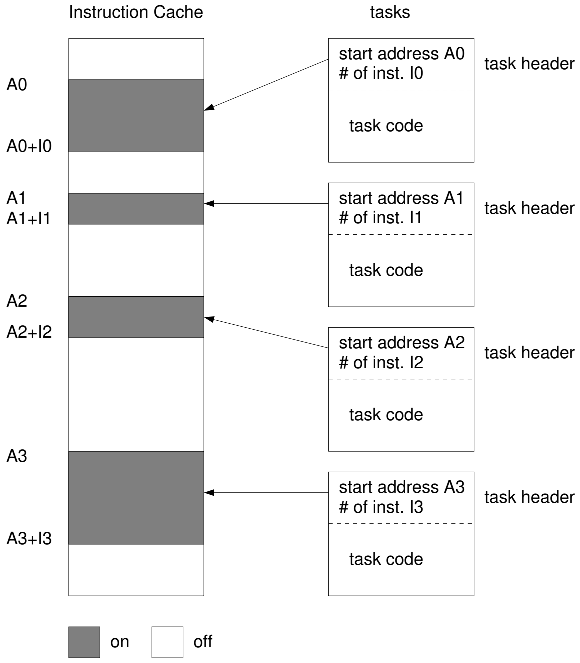

Figure 2. Basic Idea of Power Reduction Using Compiler Techniques in an Instruction Cache. The processor knows exactly which sets of an instruction cache will be used next because (1) the start address of a task is accurately predicted and (2) the task size is described in its header.

Figure 3 shows an example. Initially all instruction cache sets are turned off. When a task j is invoked at time t0 and a task j+1 is invoked at time t1 from the task dispatch unit, the addresses from A0 to A0+I3 and from A2 to A2+I2 in the instruction cache are turned on. When a task j is completed at time t2, the task is not active on another processors, and the task will not be used in the near future (i.e.,the task does not appear in a task dispatch queue), the instruction cache sets from A0 to A0+I3 for the task j are turned off. At time t3 when a task j+1 completes, the cache sets from A2 to A2+I2 are not turned off because the other processor (processor 3) is using the same start address for a task j+3. When a task j+2 completes at time t4, the cache sets from A3 to A3+I3 are not turned off because the same task is again in the task dispatch queue.

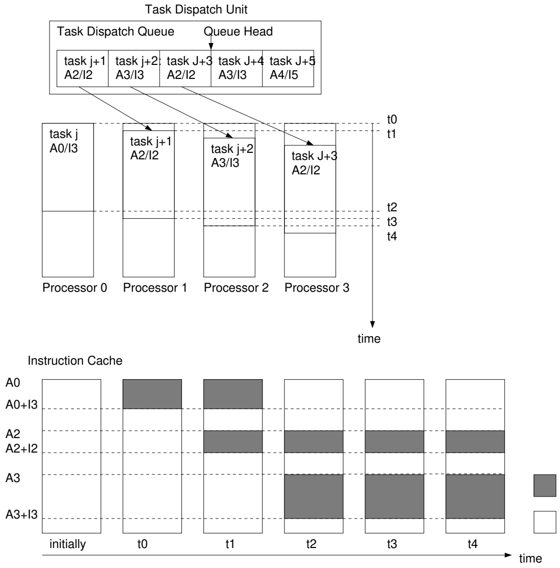

Figure 3. Example of the Proposed Compiler Techniques for Energy Savings. A task dispatch unit turns on and off the cache sets which are used for the tasks in the task dispatch queue using two types of information: the predicted target address by an inter-task predictor and the number of instructions by a compiler. This scheme turns on only these cache sets that are used by active tasks.

3.2 Hardware Extension

We assume two new hardware capabilities for our scheme: (1) the hardware can turn on and off the instruction cache set by set, and (2) the task dispatch unit is able to add the task size to the start address and turn on the needed I-cache sets in this address range. Our scheme increases the hardware complexity slightly, but results is substantial energy savings. We have added these features in the Multiscalar/CIP simulator.

4 Performance Analysis

4.1 Experimental Setup

Table 2 shows the parameters used for the performance measurement. There are four cache components on a processor: ARB, task, instruction, and data caches. The ARB cache is a speculative cache to resolve the memory dependencies at run time. We used two SPECint95 (`COMPRESS` and `GO`) and three SPECfp95 benchmarks (`SWIM`, `HYDRO2D`, and `FPPPP`) for our performance evaluation. We used the `test` data set for `COMPRESS`, `SWIM`, and `HYDRO2D`, and `train` data set for `GO` and `FPPPP` provided with the SPEC benchmarks.

4.2 Overall Performance

Figure 4 shows the overall performance of the proposed CIP scheme and the L0-scheme reported in [1] with 256 bytes L0-cache and a block size of 8 bytes. We directly quoted the results given in [1] for the L0-scheme. In `SWIM` and `HYDRO2D`, the L0-scheme reduces more power than the CIP scheme, but in `FPPPP`, `COMPRESS` and `GO`, the CIP scheme performs better. In scientific codes that take most time in loop execution such as `SWIM` and `HYDRO2D`, the L0-scheme is better than the CIP scheme because it is easy to find the most frequently used basic blocks. However, in applications with more complex structure and significant execution time in non-loop regions, the CIP scheme performs better.

Our scheme can incur three types of overheads: One is that, because we increase the size of the task header, the fetch and decode of a task header takes slightly longer. Another overhead results from task misprediction. When the task is mispredicted, unused sets of the I-cache are turned on. The correct cache sets are then only turned on with some delay, which increases the cache misses (access of a turned-off cache set results in a miss). Table 3 shows the average access latency and I-cache hit ratio without power saving and with our power saving scheme. The average access latency increases a little, and the hit ratio decreases, except in `SWIM`. Even if the hit ratio decreases in `SWIM`, other factors affect the architectural behavior, which results in faster access time. It shows that the proposed scheme performs very well. The third overhead is that if the a task is reused repeatedly with a large distance in execution time, our scheme will turn off the cache between uses and hence lose the cache content. In contrast, a processor without power saving may retain some of the instructions of this task in the I-cache. Our measurements in Figure 4 show that these overheads result in very little execution time penalty.

The execution time overhead of the L0-scheme occurs when instructions are transferred to the L0-cache, as a result of an L0-cache miss. The performance of this scheme is highly dependent on the accuracy of the branch prediction. It should be noted that the accuracy of inter-task prediction in our scheme is usually higher than the normal branch prediction used widely in modern processors. This is because the number of conditional branches in inter-task prediction is less in most cases.

In the following subsections, we will discuss the per-

Table 2. Simulation Parameters on Multiscalar.

Components		Parameters
Processor		4 processors, and each has 2-way superscalar.
Cache	ARB	4 banks, full-associative, 32 word/block. (Total 512K).
	Task	1 bank, 512 sets, 2-way, 16 word/block, LRU. (Total 64K).
	Instruction	4 banks, 256 sets, 2-way, 8 word/block, LRU. (Total 64K).
	Data L1	4 banks, 256 sets, 2-way, 8 word/block, LRU.(Total 64K).
Network	ARB	Crossbar, one read port and one write port per bank.
	Cache Off-chip	Bus, one read port and one write port per bank.

Table 3. Average Access Latency (Cycles) and Hit Ratio (%) to an Instruction Cache on Multiscalar. The average access latency to the instruction cache increases slightly except in `SWIM`, where even if the cache miss ratio increases, the average access latency decreases.

Benchmarks	Without Power Saving Scheme		With Power Saving Scheme	
	Access Latency	Hit Ratio	Access Latency	Hit Ratio
`SWIM`	1.14	100.0	1.12	99.9
`HYDRO2D`	1.11	100.0	1.13	99.9
`FPPPP`	2.11	89.4	2.32	86.6
`COMPRESS`	1.07	99.9	1.10	98.5
`GO`	1.31	97.9	1.53	92.8

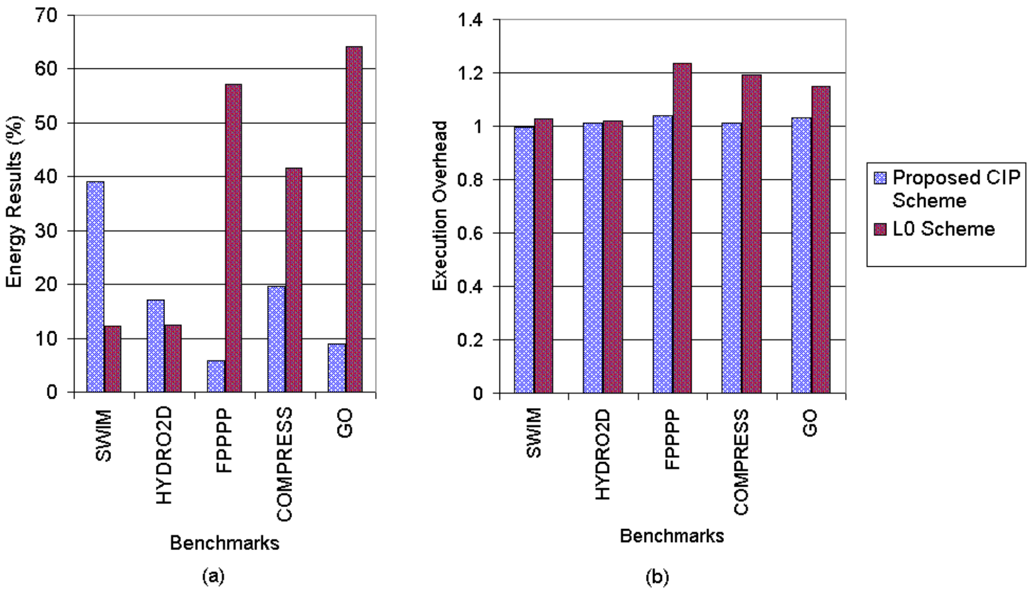

Figure 4. Performance of the New Power Saving Techniques. (a) Power Savings. (b) Execution Overhead. A 20% energy result means that we use only 20% of the power in an instruction cache. We directly quoted the results given in [8]. The execution overhead of the CIP scheme is the ratio of execution time with power saving techniques to the execution time without power saving techniques.

formance of each benchmark in more detail.

4.3 `COMPRESS` and `GO`

In `COMPRESS` and `GO`, the performance of our CIP scheme is better than that of the L0-scheme in both power reduction and execution overhead. Figure 5 (a) and (b) show the usage of the cache sets in these benchmarks. About half of the instruction cache is unused in `COMPRESS`, and in both benchmarks the I-cache access patterns are irregular. In non-numeric code with irregular access patterns, it is not easy to find frequently used basic blocks, such as loops. The performance of the CIP scheme is relatively independent of application characteristics, whereas the performance of the L0-scheme depends on the presence of identifiable, time-consuming loops. The execution overhead of the L0-scheme is very high because of the high branch misprediction, which is about 10% in `COMPRESS` and 26% in `GO` [1].

4.4 `FPPPP`

Similarly, in `FPPPP`, the performance of the CIP scheme is better than that of the L0-scheme. Figure 5 (c) shows that some code sections are executed repeatedly, but there are also irregular access patterns shown as several peaks in the figure. The benchmark `FPPPP` spends much execution time in non-loop regions, and it has high branch misprediction of about 7%. It results in less power savings and high execution overhead in the L0-scheme.

4.5 `SWIM` and `HYDRO2D`

In `SWIM` and `HYDRO2D`, the power saving of the proposed CIP scheme is less than the L0-scheme, but the CIP scheme has less execution time overhead. The reason is that, in our current speculative code generation scheme, calls to library routines are part of the calling tasks, although the library code is statically linked at the end of the application body. The called library is not described in the task header and, consequently, the task predictor does not know that the execution branches to it. When the library is called for the first time, the instruction fetches cause cache misses, as a result of which the cache sets for the library code are turned on. For similar reasons, when the execution of the library is completed, the accessed cache sets are not turned off. They remain turned on until a future task uses the same cache sets, and then turns them off. In `SWIM`, the most time-consuming loop calls hypergeometric functions, which results in less power savings in our CIP scheme. Also, `HYDRO2D` uses mathematical libraries and input functions. Figure 5 (d) and (e) show that many cache sets are turned on during all execution time, where the cache sets are used for the libraries.

5 Conclusion

We have presented a new method for energy saving in instruction caches of speculative parallel architectures. Our compiler-assisted I-cache prediction scheme is based on both compiler techniques and hardware capabilities to turn on and off individual sets of the I-cache. We have shown that the new scheme can result in up to 90% I-cache energy savings in general purpose programs without significant execution overhead. The applicability to general-purpose programs is a distinguishing feature of the new scheme. It contrasts with related approaches that have focused on programs with regular loop structures. Loop-oriented schemes may lead to higher savings in programs that exhibit certain regular patterns. A low-power processor may use the presented method in combination with such schemes.

We have focused our work on an emerging class of machines: speculative parallel microarchitectures. The presented methods grew naturally out of our compiler work with these architectures. While we believe this is one of the most interesting future classes of target machines, our scheme can easily be extended to conventional processors: Because conventional processors do not have a task concept, we introduce *a virtual task* as the aggregation of several basic blocks. In most cases the basic blocks in the same virtual task are executed under the same control flow condition. For example, a loop can be one virtual task. We augment the code to specify the number of instructions before branches that jump to out of the current virtual task. In order to implement this scheme, we need extensive control flow analysis. Also profiling can be used. In ongoing work we are developing such extensions.

We have addressed one of several important problems towards low-power processor design, which can be used in applications such as mobile computing devices and field appliances. Complementing our I-cache energy reduction techniques, we need to develop methods for reducing the power consumption of data caches. Doing so is another objective of our ongoing work.

References

[1] N. Bellas, I. Hajj, and C. Polychronopoulis. Using dynamic cache management techniques to reduce energy in a high-performance processor. *Proceedings 1999 in-*