An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors[*]

I. Kadayif
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kadayif@cse.psu.edu

M. Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

U. Sezer
ECE Department
University of Wisconsin
Madison, WI 53706
sezer@ece.wisc.edu

ABSTRACT

With energy consumption becoming one of the first-class optimization parameters in computer system design, compilation techniques that consider performance and energy simultaneously are expected to play a central role. In particular, compiling a given application code under performance and energy constraints is becoming an important problem. In this paper, we focus on an on-chip multiprocessor architecture and present a parallelization strategy based on integer linear programming. Given an array-intensive application, our optimization strategy determines the number of processors to be used in executing each nest based on the objective function and additional compilation constraints provided by the user. Our initial experience with this strategy shows that it is very successful in optimizing array-intensive applications on on-chip multiprocessors under energy and performance constraints.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Design, Experimentation, Performance

Keywords

Embedded Systems, Loop-Level Parallelism, Constraint-Based Compilation

1. INTRODUCTION

System-on-a-Chip architectures are increasingly employed to solve a diverse spectrum of problems in embedded and mobile systems domain. For example, GVPP [1], a system-on-a-chip architecture manufactured by a French company, executes 20 billion instructions per second and models the human perceptual process at the hardware level by mimicking the separate temporal and spatial functions of the eye-to-brain system. GVPP sees its environment as a stream of histograms regarding the location and velocity of objects. Those objects could be the white lines on a highway, the football in a televised game or the annotated movement of enemy ground forces from satellite telemetry. The system is also capable of learning-in-place to solve a variety of pattern recognition problems. It has automatic normalization for varying object size, orientation, and lighting conditions, and can function in daylight or darkness.

As the applications loaded into system-on-a-chip architectures become more and more complex, it is extremely important to have sufficient compute power on the chip. One way of achieving this to put multiple processor cores in a single chip. This strategy has advantages over an alternate strategy which puts a more powerful and complex processor in the chip. First, the design of an on-chip multiprocessor composed of multiple simple processor cores is simpler than that of a complex single processor system [8, 13]. Note that this simplicity also helps reduce the time spent in verification and validation [12]. Second, an on-chip multiprocessor is expected to result in better utilization of the silicon space. The extra logic that would be spent on register renaming, instruction wake-up, and register bypass on a complex single processor can be spent for providing higher bandwidth on an on-chip multiprocessor. Third, the on-chip multiprocessor architecture can exploit loop-level parallelism at the software level in array-intensive applications. In contrast, a complex single processor architecture needs to convert loop-level parallelism to instruction-level parallelism at runtime (that is, dynamically) using sophisticated (and power-hungry) strategies. During this process, some loss in parallelism is inevitable.

An on-chip multiprocessor improves execution time of applications using on-chip parallelism. An application program can be made to run faster by distributing the work it does over multiple processors on the on-chip multiprocessor. There is always some part of the program's logic that has to be executed serially, by a single processor; however, in many applications, it is possible to parallelize some parts of the program code. Suppose, for example, that there is one loop in the code where the program spends 50% of its execution time. If the iterations of this loop can be divided across two processors so that half of them are done in one processor while the other half are done at the same time in a different processor, the whole loop can be finished in half the time, resulting in a 25% reduction in execution time. While this argument favors increasing the number of processors as much as possible, there is a limit beyond which using more processors might actually degrade performance. This is because in many cases parallelizing a loop entails interprocessor communication. Increasing the number of processors in general increases the frequency and volume of this communication. After a specific number of processors is reached, increasing the number of processors further increases communication costs so significantly that additional benefits due to more parallelism may not be able to offset this.

The preceding discussion indicates that even one focuses only on performance, selecting the number of processors to use in parallelizing loop nests may not be trivial. When we consider multiple objective functions at the same time, the problem becomes even

[*]This work was supported in part by NSF Career Award #0093082.

more difficult to address. Suppose, for example, that we would like to minimize the energy-delay product of a given loop using parallelization. To decide the number of processors to use, we need to evaluate the impact of increasing the number of processors on both energy consumption and execution cycles. If using more processors does not bring significant reductions in execution time, the energy-delay product may suffer as using more processors means that more processors should be powered on. Evaluating the impact of increasing the number of processors on both energy and performance is extremely difficult if one restricts itself only to static (compile-time available) information. Finally, the possibility that each loop in a given application may demand a different number of processors to generate the best results makes the overall parallelization problem highly complex.

In this paper, we focus on a constraint-based parallelization problem and propose an integer linear programming (ILP) based strategy to select the number of processors to use in parallelizing each loop nest in an application code to be run on an on-chip multiprocessor. Our strategy has two components. First, there is a profiling phase, where we run (simulate) each loop with different number of processors and collect performance and energy data. In the second phase, called the optimization phase, we formulate the problem of selecting a processor size for each loop as an integer linear programming (ILP) problem and solve it. Our ILP-based approach can take into account multiple constraints that involve execution time and energy consumption on different hardware components. To the best of our knowledge, this is the first constraint-based application parallelization study for on-chip multiprocessors.

The remainder of this paper is organized as follows. Section 2 gives an overview of the on-chip multiprocessor architecture considered in this paper. Section 3 demonstrates why parallelization for minimum execution time and parallelization for minimum energy consumption demand different strategies. Section 4 presents details of our constraint-based parallelization strategy and Section 5 presents experimental results. Finally, Section 6 concludes the paper by summarizing our major contributions.

2. ON-CHIP MULTIPROCESSOR

In this paper, we focus on an on-chip multiprocessor architecture. This architecture contains multiple processors (each with its own instruction and data caches), a shared on-chip SRAM memory, and an interprocessor synchronization logic. This is a shared memory architecture; that is, all interprocessor communication occurs through reading from and writing into the shared SRAM memory. A bus-based on-chip interconnect is used to perform interprocessor synchronization. This synchronization is necessary for the processors to get synchronized at the beginning and end of each nest they execute.

As mentioned in the previous section, an on-chip multiprocessor architecture has some advantages compared to superscalar and VLIW machines. Maybe the most important of these is the simplicity and independence of the processors. The processors we assume in this study are single-issue, five-stage pipelined architectures without any branch prediction or data speculation logic. This brings an important side-advantage in terms of execution time predictability as it is easier to predict execution time with simple processors without sophisticated prediction/speculation logic. Also, each processor can operate independently from each other and processors engage in synchronization only to maintain data integrity during parallel execution.

On-chip multiprocessors are very suitable for array-intensive embedded applications. Many array-intensive embedded applications are composed of multiple independent loop nests that operate on large, multidimensional arrays of signals. An important issue in such applications is to map the application code to the embedded architecture in question. Note that, in the on-chip multiprocessor case, this is relatively simple as, in the source-code level, we can parallelize each loop nest and distribute iterations across the processors in the on-chip multiprocessor. If the loop in question is parallelizable, high execution time benefits can be achieved using this strategy. In contrast, a superscalar or a VLIW architecture should map parallel loop iterations to parallel (and sometimes irregular)

Simulation Parameter	**Value**
Processor Speed	400MHz
Number of Processors	8
Instruction Cache	4KB 2-way associative 32 byte blocks
Data Cache	4KB 2-way associative 32 byte blocks
Memory	1 MB
Cache Dynamic Energy Consumption	0.6 nJ
Memory Dynamic Energy Consumption	1.17 nJ
Leakage Energy Consumption for 32 bytes	
Active State	4.49 pJ
Sleep State	0.92 pJ
Resynchronization Time	20 msec

Figure 1: Simulation parameters.

functional units, which may not form a good match for the application structure.

Figure 1 gives the simulation parameters used in this paper. We have used Cacti [17] to obtain cache and memory dynamic energy values. We assume that the leakage energy per cycle for 4KB SRAM is equal to the dynamic energy consumed per access to a 32 byte data from that SRAM. This assumption tries to capture the anticipated importance of leakage energy in the future as leakage becomes the dominant part of energy consumption for 0.10 micron (and below) technologies for the typical internal junction temperatures in a chip [3]. Since not all processors are used in executing a given loop nest, the unused processors and their instruction and data caches can be placed into a power-down (sleep) mode (state). In the power-down state, the processor and caches consume only a small percentage of their original (per cycle) leakage energy. However, when a processor and its data and instruction caches in the sleep state are needed, they need to be reactivated (resynchronized). This resynchronization costs extra execution cycles as well as extra energy consumption. Note also that since the SRAM main memory is shared between processors, it is never placed into power-down state. In this paper, we use the terms ‘number of processors’ and ‘processor size’ interchangeably.

It should be mentioned that on-chip multiprocessors are gaining popularity in commercial world as well. A number of architectures such as IBM’s Power4, Sun’s MAJC-5200, and MP98 contain multiple processors (currently between two and four) on a single die [9, 11, 5, 4].

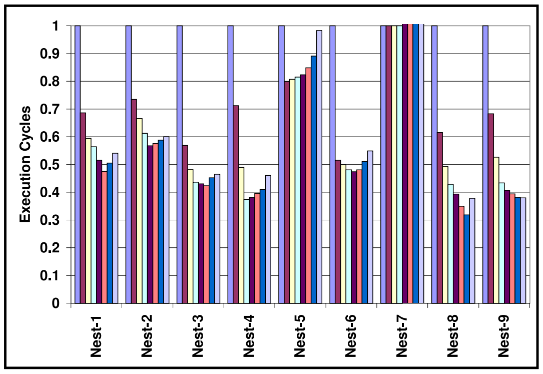
Tuning processor’s resources according to the needs of application has been studied by previous research in the context of superscalar processors [6, 10, 2]. In contrast, in this work, we focus on optimizing a given on-chip multiprocessor application in a constrained-based environment using integer linear programming.

3. IMPACT OF ON-CHIP PARALLELIZATION ON ENERGY

To see whether parallelization for energy and parallelization for performance are the same thing or not, we conducted an initial set of experiments where we run each nest of each code in our experimental suite using different number of processors (ranging from 1 to 8), and measured the energy consumption and execution cycles. Since the trends for different benchmarks were similar, we focus here only on tomcatv, one of our array-intensive benchmarks.

Figure 2 gives the execution time and energy consumption for each of the nine nests of this benchmark. For each nest, the eight bars from left to right correspond to different number of processors from 1 to 8. Also each bar represents a result *normalized* with respect to the single processor case (that is, the energy consumption or execution cycles when the nest is executed on a single processor). The details of our benchmarks and simulation environment will be given in a later section.

From these graphs, we can make several observations. First, en-



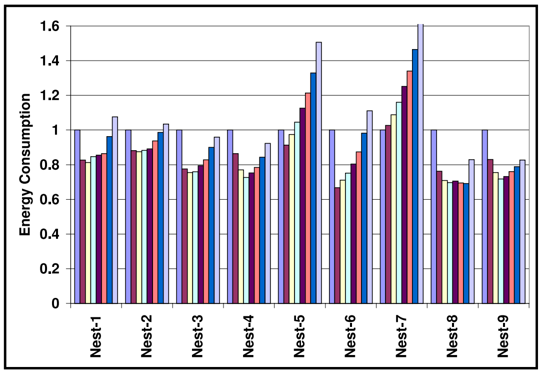


Figure 2: Normalized execution cycles and energy consumptions for tomcatv with different processor sizes.

ergy and performance trends are different from each other. This is because in many cases increasing the number of processors beyond a certain point may continue to improve performance, but the extra energy to power on the additional processor and its caches may offset any potential energy benefits coming from the reduced execution cycles. Second, for a given nest, increasing the number of processors does not always reduce execution cycles. This occurs because there is an extra performance cost of parallelizing a loop, which includes spawning parallel threads, interprocessor synchronization during loop execution if there are data dependences, and synchronizing threads after the loop execution. If the loop does not have sufficient number of iterations to justify the use of a certain number of processors, this extra cost can dominate the performance behavior. Third, the best number of processors from energy or performance perspectives depend on the loop nest in question.

4. OUR APPROACH

Our approach has two phases: profiling phase and optimization phase. In the profiling phase, we run each nest of the application being optimized for each possible processor size and record the energy consumption and execution time. These data are then fed to the second phase where we use integer linear programming (ILP) to determine the number of processors for each nest taking into account the objective function and compilation constraints. Therefore, the number of processors that will be used for executing each nest in the final optimized code is decided at compile time. However, the processor deactivations/reactivations (that is, placing a processor and its caches into a sleep state and later transitioning them to active state) take place at runtime.

To be more specific, if we have M different nests in a given array-intensive code and N different processors in our on-chip multiprocessor, we prepare two tables (one for energy values and the other one for execution cycles) in which each nest has N entries (that is, a total of NM energy values and NM execution time values). An entry i, j in these tables denote the energy consumption (or execution cycles) when nest i is executed using j processors. While in this paper we experimented with an architecture that contains eight processors, it is not difficult to modify this strategy to work with a machine with different number of processors. After building such tables (using the energy simulator that will be discussed shortly), our approach takes into account the energy and performance constraints (given as input) and selects the best number of processors for each nest. It should be noted that the number of processors selected for two consecutive nests might be different from each other, and in switching from one processor size to another, the associated time and energy overhead has to be accounted for. To select the processors sizes for the nests, our approach formulates the problem as a zero-one ILP (ZILP) and uses a publicly-available solver [14].

Let us first make the following definitions, assuming that $1 \leq i \leq M$ and $1 \leq j \leq N$, where N is the total number of processors and M is the number of nests in the code being optimized:

- $E_{i,j}$: the estimated energy consumption for nest i when j processors are used to execute it.
- $X_{i,j}$: the estimated execution time for nest i when j processors are used to execute it.

We use zero-one integer variables $s_{i,j}$ to indicate whether j processors are selected for executing nest i or not. Specifically, if $s_{i,j}$ is 1, this means that j processors are selected to execute nest i in the final solution. Obviously,

$$\sum_{j=1}^{N} s_{i,j} = 1 \tag{1}$$

should be satisfied for each nest i. Consequently, the total energy consumption (E) and total execution time (X) for the application can be expressed as:

$$E = \sum_{i=1}^{M}\sum_{j=1}^{N} s_{i,j}E_{i,j} \tag{2}$$

$$X = \sum_{i=1}^{M}\sum_{j=1}^{N} s_{i,j}X_{i,j} \tag{3}$$

It should be noted, however, that these energy and execution time calculations do not include any overhead for changing the number of processors between nest boundaries.

In our experiments, we assumed that a fixed (constant) amount time (denoted R) is required to make an inactive processor (i.e., a processor that has been placed into the sleep state along with its caches) active. We can compute the execution time overhead as follows:

$$n_{i,j} \geq s_{i,j} - \sum_{k=j}^{N} s_{i-1,k} \tag{4}$$

$$y_i = \sum_{j=1}^{N} n_{i,j} \tag{5}$$

$$Y = R\sum_{i=2}^{M} y_i \tag{6}$$

where $2 \leq i \leq M$ and $1 \leq j \leq N$. In this formulation, $n_{i,j}$ and y_i are zero-one variables. Expression (4) restricts $n_{i,j}$ to be 1 if nest i has j active processors and nest $i-1$ has less than j active processors; otherwise, $n_{i,j}$ is restricted to be 0. Expression (5), on the other hand, sets y_i to 1 if nest i has more active processors than nest $i-1$. Note that only when nest i uses more processors than nest $i-1$ we incur a resynchronization penalty in going from nest $i-1$ to nest i. Finally, expression (6) computes the total time overhead for activating processors between nest boundaries. Note that R is constant (20 msec in our experiments).

Suppose that in going from one nest to another, we need to activate new processors. These new processors plus the ones used to execute the former nest are considered to consume leakage energy for the time period of R. The energy consumption including overheads can be computed as follows:

$$
\begin{aligned}
t_{i,j} &= \sum_{k=1}^{j} s_{i,k} && (7)\\
z_i &= \sum_{j=1}^{N} t_{i,j} - 1 && (8)\\
u_i &\geq z_{i-1} - z_i && (9)\\
U &= \sum_{i=2}^{M} u_i && (10)\\
Z &= r(U + \sum_{i=1}^{M-1} \sum_{j=1}^{N} j s_{i,j}) && (11)
\end{aligned}
$$

where $1 \leq i \leq M$ and $1 \leq j \leq N$ except for expression (9) where $2 \leq i \leq M$. Also, r is the leakage energy consumption of a single processor during the time period of R and is constant. In this formulation, $t_{i,j}$ is a zero-one variable. Expression (7) here sets $t_{i,j}$ to 1, if nest i has j or less than j active processors. Expression (8) gives the number of inactive (powered-down) processors at nest i. Expression (9) forces u_i to take 0 or a positive value and, determines the number of processors activated in going from nest $i-1$ to nest i. Expression (10) gives the total number of processors that need to be activated between nest boundaries. Finally, expression (11) gives the total leakage energy overhead for the processors activated between nest boundaries and processors already active during activation periods.

Note that, taking into account these energy and performance overheads, $X+Y$ gives the execution time including overheads and $E+Z$ gives the energy consumption including overhead energy. It should also be noticed that expressions for calculating time overhead and energy overhead are different in a sense that while we overlap the time overhead for activated processors between loop boundaries, we consider each activated processor to consume leakage energy during the activation period.

In some cases, the optimization problem involves constraints or objective function built from energy consumptions of different hardware components. One example would be minimizing main memory energy only (instead of the overall system energy) under a specific execution time and a specific overall energy consumption constraints. To capture the energy breakdown across different components, we use $E^d_{i,j}$, $E^{dc}_{i,j}$, $E^{ic}_{i,j}$, and $E^m_{i,j}$ to denote (for nest i executing using j processors) the estimated energy consumptions in datapath, data cache, instruction cache, and main memory, respectively. If these are the only hardware components of interest, we have:

$$E_{i,j} = E^d_{i,j} + E^{dc}_{i,j} + E^{ic}_{i,j} + E^m_{i,j}.$$

We can find the overall main memory (E^m), instruction cache (E^{ic}), data cache (E^{dc}), and datapath energies (E^d) as:

$$
\begin{aligned}
E^m &= \sum_i \sum_j E^m_{i,j} s_{i,j}\\
E^{ic} &= \sum_i \sum_j E^{ic}_{i,j} s_{i,j}\\
E^{dc} &= \sum_i \sum_j E^{dc}_{i,j} s_{i,j}\\
E^d &= \sum_i \sum_j E^d_{i,j} s_{i,j}.
\end{aligned}
$$

5. EXPERIMENTS AND RESULTS

We made experiments with several array-intensive benchmarks. The important features of our benchmarks are listed in Figure 3. Among these codes, `3step-log`, `full-search`, `hier`, and `par allel-hier` are different implementations for motion estimation. The remaining codes are representative array-intensive benchmarks from Spec and Perfect Club suites. The fourth column gives execution times in msecs and the last two columns give overall dynamic and leakage energy consumptions, in millijoules, for each benchmark when they are executed on a single processor of our on-chip multiprocessor. As shown in the table, leakage energy dominates dynamic energy since whole memory is active regardless of the number of active processors, and at each cycle, active data cache, instruction cache and memory consume leakage energy even there are no accesses to them.

Figure 4 shows the structure of our optimization system. The input code is first optimized using classical data locality optimizations as well as loop transformations that enhance loop-level parallelism (e.g., loop skewing) [16]. Then, this optimized code is fed to our simulator (see below). The simulator simulates each nest using all possible processor sizes and records energy consumption (both total and component-wise) and execution time for each nest. After that, this information along with the code itself is fed to the parallelization module (shown as Parallelizer in the figure). This module builds up ILP formulation (in a form suitable for the solver) and passes it to the ILP solver. The solver also takes the compilation constraints and the objective function as input. The parallelization module then obtains the solution from the solver and parallelizes the application (i.e., each nest) accordingly. If it is not possible to parallelize the application under given constraints and objective function, an error message is signaled. If desired, the user can relax some constraints and retry compilation.

Our simulation methodology is as follows. We simulate an on-chip multiprocessor architecture that contains 8 identical single-issue processors. Our simulation model has four parts: datapath, instruction and data caches, (shared) main memory, and interprocessor synchronization. For each processor, datapath energy is obtained using a cycle-accurate energy simulator developed for a five-stage, single-issue pipelined processor [15]. This simulator employs transition-sensitive energy models for datapath components such as functional units and register file. It has been validated using a commercial low-power microprocessor and its accuracy has been shown to be within 10% (of real current measurements). For cache memories and main memory (in our case, an SRAM), we used the analytical energy model proposed by Kamble and Ghose [7]. Since in our on-chip multiprocessor interprocessor synchronization is through a bus-based mechanism, we modeled its energy consumption as that of placing 8 bits into the interconnect (one for each processor) and the associated control register update (to set/reset the synchronization bit). Note that processors share data through main memory; that is, there is no extra interconnect for explicit interprocessor data communication. All the results presented in this section were obtained using the simulation parameters given in Figure 1. Also, all energy values reported in this paper are based on parameters for 0.10 micron, 1V technology.

To see the energy and performance benefits of our approach, for each benchmark, we performed experiments with two different versions. The first version is the original (unoptimized) code that uses the maximum number of processors (8) for each nest. The second version, is the optimized code where the number of processors for each nest is selected using the ILP-based strategy detailed in the previous section. Apart from the number of processors used for parallelizing loops, both the original and optimized codes have exactly the same structure (i.e., both of them are loop and data transformed for the best execution). After showing that the optimized codes exhibit much better energy behavior than their unoptimized counterparts, we demonstrate how our strategy can be used for compiling a given application under different energy/performance constraints.

Our presentation is in three parts. First, we give overall energy and performance results when our strategy is used for optimizing the total energy and performance. In our context, total energy is the sum of energies expended in datapath, instruction and data caches, main memory, and interprocessor synchronization bus. These results are given in Figure 7. The first graph in this figure gives the execution time of the optimized code as a fraction of the corresponding original code when the latter uses all 8 processors for all the nests. We see that the savings are modest (6.9% on the average). The second graph shows the overall energy consumption of the optimized code as a fraction of the original code when the latter uses all 8 processors for all the nests. We observe that the energy savings are much higher than performance savings. This is because, as

Benchmark	Input Size (KB)	Number of Nests	Exec. Time	Dynamic Energy	Leakage Energy
3step-log	814.16	3	1271.8	139.0	1709.3
adi	242.60	2	1187.6	129.5	1596.1
aps	548.92	3	764.4	98.8	1019.2
bmcm	525.44	4	1236.2	128.7	1661.4
btrix	212.04	7	3076.3	418.7	4134.7
eflux	780.30	2	268.8	42.5	361.3
full-search	814.16	3	4546.1	464.0	6109.9
hier	814.16	7	6456.7	529.8	8677.7
lms	320.00	4	584.2	106.0	785.2
n-real-upd.	80.00	3	2640.9	456.0	3549.4
par.-hier	814.16	5	5784.4	746.4	7774.4
tomcatv	633.60	9	2534.2	359.6	3406.0
tsf	468.18	4	2934.0	351.8	3943.3

Figure 3: Array-intensive benchmarks.

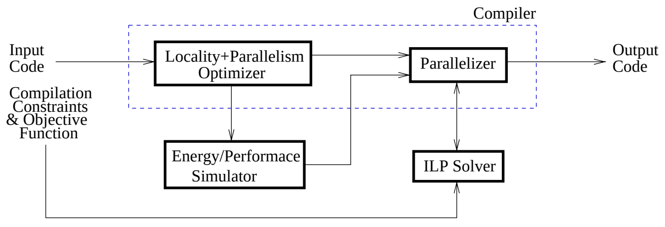
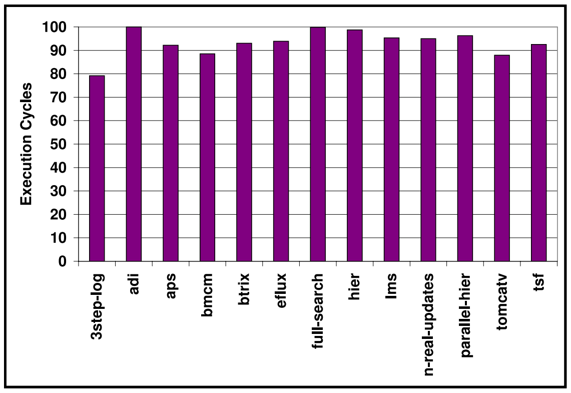

Compiler
Input Code
Locality+Parallelism Optimizer
Parallelizer
Output Code
Compilation Constraints & Objective Function
Energy/Performace Simulator
ILP Solver

Figure 4: Our optimization system.

can be seen from the tomcatv example in Figure 2, optimizing (i.e., generating the best result) for performance generally requires large number of processors than required for optimizing for energy consumption. The largest energy savings in Figure 7 occur with `aps` (35.3%), `lms` (52.1%), and `tsf` (54.4%). This is because each of these codes contains a small number of costly nests that demand a small number of processors for the best energy behavior. The results presented in Figure 7 also reveal that it is not a good idea from the energy perspective to work always with the maximum number of processors available. In other words, for the best energy results, the number of processors should be changed adaptively.

In the second part of our presentation, we focus on tomcatv and give detailed results when this application is compiled under different energy and performance constraints. Let us assume for the moment that the overhead of dynamically changing the number of active processors is zero. We consider different compilation strategies whose objective functions and performance/energy constraints (if any) are given in the second and third columns, respectively, of Figure 5. First, let us focus on two simple cases: Case-I and Case-II. In Case-I, we try to minimize execution time; note that this is the classical objective of a performance-oriented compilation strategy. We see from columns four through twelve that in this case the average number of (selected) processors per nest is 4.67. Case-II represents the other end of the spectrum where we strive for minimizing the energy consumption (without any performance concerns). The average number of processors per nest is 3.22. The last two columns in Figure 5 give the energy consumption (in millijoules) and execution time (in msecs) for each scenario. We observe that optimizing only for performance results in an energy consumption which is 11.5% higher than that of the best energy-optimized version. Similarly, optimizing only for energy consumption leads to an execution time which is 14.5% higher than that of the best performance-optimized version. These results clearly show that parallelization for energy and performance are different objectives.

Let us now focus on other compilation strategies. Case-III and Case-IV correspond to minimizing energy consumption under performance constraints. We note that the processor sizes selected in these cases are different from those selected for Case-II. The average number of processors per nest are 3.88 and 4.11 for Case-III and Case-IV; that is, including a performance constraint in the compilation increases the average number of processors selected. In addition, we see that a more stringent performance constraint (e.g., $X \leq 925$) results in higher number of processors. Case-V corresponds to optimizing execution time under energy constraint. Note that, in this case, the number of processors selected for some nests is different from the corresponding numbers in Case-I. To sum up, when we consider Cases III, IV, and V, we see that including constraints during compilation can change the number of processor selected for executing nests. Note also that without an ILP-based framework such as the one presented in this paper, it would be extremely difficult to decide on the number of processors for each nest when a compilation scenario such as Case-V, Case-II, or Case-III is encountered.



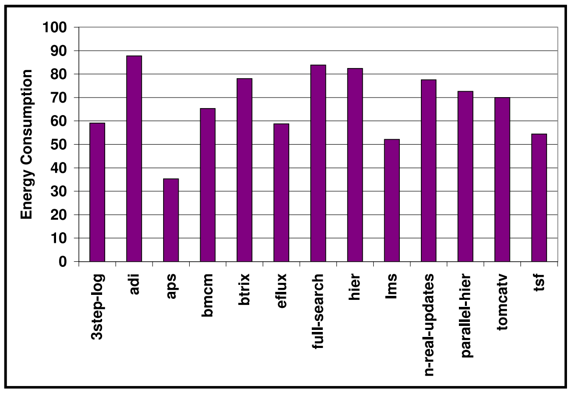


Figure 7: Normalized execution cycles and energy consumptions.

Up to this point, we have assumed that the overhead of dynamically changing the number of processors between nests is zero. The remaining cases in Figure 5 correspond to scenarios with non-zero energy and performance overhead for reactivating an inactive processor. Specifically, we assumed a resynchronization penalty of 20 msec (which is a highly conservative [large] value even for powerful leakage optimization techniques such as power supply gating [3]) during which all processors (except the ones in low power) are assumed to consume leakage energy. We see from Cases VI, VII, and VIII that when the overhead is taken into account, the ILP solver tends to keep the number of processors the same between the neighboring nests as much as possible. We also see that even in these cases, optimizing for performance and optimizing for energy generate different number of processors for most of the nests.

So far, we focused on overall energy consumption without considering individual hardware components. In our last set of experiments, we investigate how targeting energy consumptions of different components might affect the selected number of processors. As before, we focus only on tomcatv. Our observations for other benchmarks are similar. Figure 6 gives the number of loops

Case	Objective Function	Compilation Constraints	Selected Number of Processors									Energy Consumption	Execution Time
			n1	n2	n3	n4	n5	n6	n7	n8	n9		
Case-I	minimize X	none	6	5	6	2	2	5	1	7	8	3065	908
Case-II	minimize E	none	3	3	3	4	2	2	1	7	4	2750	1040
Case-III	minimize E	$X \leq 975$	5	5	4	4	2	2	1	7	4	2814	974
Case-IV	minimize E	$X \leq 925$	6	3	4	4	2	4	1	7	7	2944	921
Case-V	minimize X	$E \leq 2975$	6	5	5	4	2	4	1	6	8	2974	915
Case-VI	minimize $E+Z$	$X+Y \leq 975$	6	4	4	4	2	2	1	4	4	2999	975
Case-VII	minimize $E+Z$	$X+Y \leq 925$	6	6	6	4	4	4	1	7	7	3253	924
Case-VIII	minimize $X+Y$	$E+Z \leq 2975$	6	3	3	3	2	2	1	3	3	2974	998

Figure 5: Different compilation strategies and the selected number of processors for each nest (tomcatv).

Case	Objective Function	Compilation Constraints	Selected Number of Processors									Energy Consumption			
			n1	n2	n3	n4	n5	n6	n7	n8	n9	E	E^m	E^{ic}	E^{dc}
Case-II	minimize E	none	3	2	2	2	2	2	1	3	3	2887	1025	628	446
Case-IX	minimize E^m	none	6	5	6	5	2	5	1	7	8	3354	856	899	690
Case-X	minimize E^{ic}	none	1	1	1	1	1	1	1	1	1	3765	1845	622	451
case-XI	minimize E^{dc}	none	1	1	1	1	1	1	1	1	1	3765	1845	622	451

Figure 6: Different compilation strategies and the selected number of processors for each nest (tomcatv).

selected for each nest and the corresponding overall energy consumption (E) as well as the energy consumptions in main memory (E^m), instruction cache (E^{ic}), and data cache (E^{dc}) when different objective functions are used (Case-II is repeated here for comparison purposes; it differs from the one in Figure 5 as E here includes overhead energy as well). We make several observations from these results. First, optimizing energy consumption of a specific component generates very different results than optimizing overall (total) energy consumption. For example, when the objective function is minimizing the main memory energy consumption (that is, Case-IX), the resulting overall energy consumption becomes 16.2% higher than the case when we try to minimize overall energy. This is despite the fact that Case-IX reduces the main memory energy by 16.5% with respect to Case-II. Second, similar observations can be made when we consider Case-X and Case-XI. Third, it is also interesting to see that if one targets only instruction cache energy or data cache energy, the ILP solver favors single processor execution for each nest. This is because in a single processor execution seven data caches and seven instruction caches can be placed in the sleep state saving significant leakage energy. As can be seen from Figure 6, however, such a single processor execution strategy increases the overall energy consumption by more than 30% compared to Case-II. It should also be mentioned, however, that if main memory resides in a different chip, then objective functions such as 'minimize E^m' or 'minimize $E^{ic}+E^{dc}+E^{d}$' might be more desirable. These example scenarios show how flexible our technique is in compiling a given code under constraints.

6. CONCLUSIONS

This paper presented an integer linear programming (ILP) based strategy for compiling a given array-intensive application in an on-chip multiprocessor under energy/performance constraints. Our strategy has two parts: profiling and ILP formulation. At the end of the optimization process, the compiler determines the most suitable number of processors that will be used in executing each nest in the code. Our preliminary results are encouraging, indicating up to 54% savings in energy compared to a static scheme that uses the maximum number of available processors for each nest. We also demonstrated that our approach can optimize a given application targeting energy consumptions of individual hardware components.

7. REFERENCES

[1] System-On-A-Chip Mimics Human Vision. http://content.techweb.com/wire/story/ TWB20000309S0002

[2] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In Proc. *the 28th Annual International Symposium on Computer Architecture,* pages 218–229, 2001.

[3] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[4] Chip Multiprocessing. http://industry.java.sun.com/javanews/stories/print/0,1797,32080,00.html

[5] Chip Multiprocessing. *ITWorld.Com,* http://www.itworld.com/Comp/1092/CWSTO54343/

[6] T. M. Conte, K.N. Menezes, S. W. Sathaye, and M. C. Toburen. System-level power consumption modeling and tradeoff analysis techniques for superscalar processor design. *IEEE Transactions on VLSI Systems,* vol 8, no. 2, April 2000.

[7] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *International Symposium on Low Power Electronics and Design,* August 1997.

[8] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multi-threading. *IEEE Transactions on Computers, Special Issue on Multi-threaded Architecture,* September 1999.

[9] MAJC-5200. http://www.sun.com/microelectronics/MAJC/5200wp.html

[10] D. Marculescu. Profile-driven code execution for low power dissipation. In Proc. *the International Symposium on Low Power Electronics and Design,* pp. 253–255, 2000.

[11] MP98: a mobile processor. http://www.labs.nec.co.jp/MP98/top-e.htm.

[12] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluating alternatives for a multiprocessor microprocessor. In Proc. *the 23rd Intl. Symp. on Computer Architecture,* pp. 66–77, Philadelphia, PA, 1996.

[13] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single chip multiprocessor. In Proc. *the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 2–11.

[14] H. Schwab. *lp_solve Mixed Integer Linear Program Solver,* ftp://ftp.es.ele.tue.nl/pub/lp_solve/

[15] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *International Symposium on Computer Architecture,* June 2000.

[16] M. Wolfe. *High Performance Compilers for Parallel Computing,* Addison-Wesley Publishing Company, 1996.

[17] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits,* May 1996.