Reducing Dynamic Compilation Overhead by Overlapping Compilation and Execution *

P. Unnikrishnan, M. Kandemir, and F. Li
Computer Science and Engineering Department
Pennsylvania State University
e-mail: {unnikris,kandemir,feli}@cse.psu.edu

Abstract— An important problem in executing applications in energy-sensitive embedded environments is to tune their behavior based on dynamic variations in energy constraints. One option for achieving this is dynamic compilation — compiling code fragments on the fly to adapt to changing energy demands. While dynamic compilation can be very beneficial in many embedded environments where multiple criteria need to be satisfied during execution, it can also incur a significant performance overhead since compilation takes place at runtime. The goal in this work is to reduce this performance overhead of dynamic compilation by overlapping it with application execution. Specifically, provided that we have available hardware resources to perform dynamic compilation concurrently with application execution, our approach compiles the next code fragment to be executed while we are executing the current code fragment. The experimental results from our implementation indicate significant savings in execution times. Our experimental results also indicate that the proposed strategy performs consistently well under different parameters.

I. Introduction

Dynamic compilation and linking is an important technique for optimizing applications while they are executing. Most of the prior work in the area [1, 3, 4, 7] focused on using dynamic compilation for implementing performance-oriented compiler optimizations at runtime. For example, the value of a program variable may not be known statically (that is, at compile time), but once it is known at runtime it may enable several optimizations (e.g., constant propagation). Similarly, a code region turns out to be executed very frequently at runtime and thus deserves a more sophisticated compilation at runtime (anticipating frequent future executions). A recent study [11] also demonstrates how dynamic compilation and linking can be used for energy adaptation in battery-operated embedded environments; that is, when energy constraints (e.g., battery level) change, the application is recompiled (dynamically) to generate a more energy-efficient version. For example, when battery is low, some memory banks in the system may need to be turned off (to save power), and being able to work with fewer number of banks may demand recompilation of the application code. In contrast, when battery power is high, the application code can make use of all memory banks available in the system – this may require another dynamic compilation.

One of the most important problems in an energy-sensitive embedded platform that employs dynamic compilation is the extra time and energy taken by the dynamic compilation process itself. This is because this performance and energy overhead directly contributes to the execution energy and time (as compilation occurs at runtime). While it may not be possible to hide the compilation energy, it may be possible to hide some portion of the compilation overhead (time) by *overlapping* it with the application execution. In other words, provided that we have available hardware resources to perform dynamic compilation concurrently with application execution, we can compile the next code fragment to be executed while we are executing the current code fragment. The code fragment in question can be a loop nest, a subroutine, or several logically-related subroutines. Overlapping dynamic compilation with application execution is termed as *compilation parallelization* in this paper (since compilation occurs parallel with application execution). It should be emphasized that, while not quantified in this paper, reducing the time spent in dynamic compilation can also be beneficial from a leakage energy consumption viewpoint (as a side effect of reduction in execution cycles).

In this paper, we propose a strategy to hide the time spent in dynamic compilation. Our strategy is based on *predicting* the next code fragment to be executed and *pre-compiling* that fragment before it is actually needed. Krintz et al [8] propose an optimization strategy to reduce the performance overhead due to dynamic compilation. Their approach mainly targets Java applications and involves maintaining a global priority queue that determines the modules to be compiled ahead of time. The efficiency of this scheme depends heavily on the strategy employed to fill the priority queue and also the time taken to fill the queue. Our strategy makes use of a global history table to predict the next code fragment to compile, and is found to be 92.37% accurate in predicting the next code fragment that will be executed. Another difference between the two studies is that our target environment is a chip multiprocessor-based energy-sensitive embedded system and our idea is applicable to different programming environments. It should be observed that an on-chip multiprocessor platform is particularly suitable for parallelizing dynamic compilation since it has multiple processor cores, one of which can perform compilation while the others can execute the application. It is known that, in many multiprocessor applications, it may not be possible to fully utilize all the available processor cores. In such cases, using one of the cores for compilation does not affect the application execution in any significant way. We implemented the proposed optimization using Dyninst [2], a post-compiler program manipulation tool, and performed experiments using several applications. Our experimental results indicate that it is possible to hide a large percentage of the compilation time (32% on the average) if one employs one processor for dynamic compilation alone. In addition, our experimental results also show that allocating more processors for dynamic compilation can

*This work is supported in part by NSF Career Award #0093082 and by a grant from GSRC.

0-7803-9451-8/06/$20.00 ©2006 IEEE.

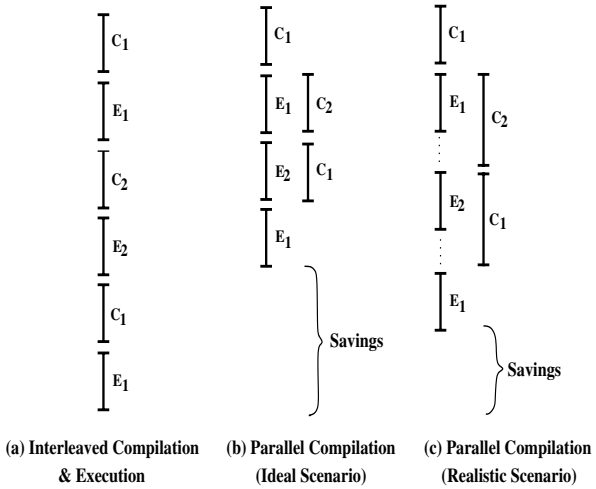

Fig. 1. Different scenarios for dynamic compilation.

be beneficial until a threshold point is reached.

The remainder of this paper is organized as follows. Section II explains our approach and discusses our implementation. Section III presents the execution model in detail and discusses history-based next module (function/nest) prediction. Section IV introduces our benchmarks and gives experimental results. Section V concludes the paper with a summary.

II. Overlapping Compilation with Application Execution

A. Approach

Our goal in this paper is to hide the time spent in dynamic compilation as much as possible. Fig. 1 helps visualize why parallelizing compilation (i.e., overlapping compilation with application execution) might be useful in practice. In this figure, C_n denotes the dynamic compilation time for module (function/subprogram or loop nest) n before it is executed. E_n denotes the execution time for module n. We consider the worst case scenario where each module has to be optimized and recompiled before it can be executed (this may occur, for example, when constraints such as remaining battery power are constantly changing). Fig. 1(a) illustrates what happens when dynamic compilation is not parallelized. Since in this case the compilation and execution are interleaved over the lifetime of the application, the overall execution time of the application can be expressed as the sum of the compilation times and execution times of each module, i.e., $(C_1 + E_1 + C_2 + E_2 + \cdots + C_n + E_n)$. It is to be noted that this is the current state-of-the-art in dynamic compilation.

In comparison, Figures 1(b) and (c) illustrate the potential benefits of overlapping compilation with application execution. In Fig. 1(b), it is assumed that we are able to hide the entire time spent in dynamic compilation (except for the first module of course). Under this assumption, the overall execution time of the application can be expressed as $(C_1 + E_1 + E_2 + \cdots + E_n)$. It should be noted however that, depending on the actual values of C_i and E_i, this ideal scenario may not be achieved all the time. Fig. 1(c) depicts a more realistic scenario, where dynamic compilation takes place in parallel with application execution, however, the compilation is not always complete in time for execution. Consequently, the application has to wait until the compilation of the module is complete. In this case, the overall execution time of the application is given by $(C_1 + E_1 + E_2 + E_3 + \cdots + E_n + TotalWaitingTime)$, where $TotalWaitingTime$ is the total time that the application has to wait for the compilation to complete. Under this scenario, the application in question incurs some compilation overhead but a significant saving can still be obtained when compared to the case in Fig. 1(a).

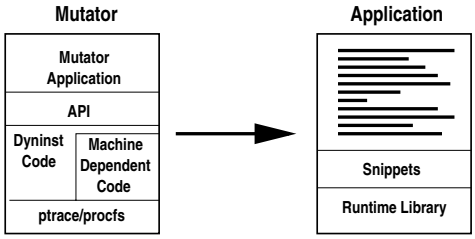

Fig. 2. Abstractions used in the Dyninst API.

One might argue at this point that instead of utilizing extra resources for dynamic compilation, it may be a better idea to employ them in normal application execution. As a concrete example, instead of using one processor for application execution and another one for dynamic compilation, it may be a better option to use both these processors for application execution (and each can do dynamic compilation on a need basis). However, such an alternative may or may not be viable, depending on the specific case at hand. In particular, some embedded applications are not amenable to thread-level parallelization. As a result, they cannot take advantage of extra processors available in the system. However, as will be demonstrated in this paper, most such applications can still take advantage of extra resources if those resources are used for dynamic compilation.

B. Implementation Details

Our dynamic compilation infrastructure is implemented using the Dyninst software from the University of Maryland [2]. Dyninst is a post-compiler program manipulation tool which provides an Application Program Interface (API) called DyninstAPI for program instrumentation. Using the DyninstAPI library, it is possible to instrument and modify application programs during execution (i.e., as they are running). DyninstAPI is itself a C++ class library which can be included and directly called from a C++ program. With this interface, a program can create a new piece of code and insert it to another program while the latter is executing. The program being modified is able to continue execution and does not need to be recompiled entirely. While, as far as the applicability of our approach is concerned, it is not very important whether dynamic compilation is needed for performance or energy reasons, our current target environment in an energy-sensitive SoC platform where we have multiple cores and some of these cores can be used for parallelizing dynamic compilation. That is, in our environment, dynamic compilation is invoked as a result of some change in energy constraints at runtime. However, our approach is oriented towards reducing the performance impact of dynamic compilation rather than its energy consumption.

The overall structure of the Dyninst API and its implementation are shown in Fig. 2 (from [2]). There are two processes, called the mutator and the application (or mutatee). The left side of the figure shows the code for the mutator process that contains calls into the Dyninst API. It also contains the code that implements the runtime compiler and the utility routines to manipulate the application process (shown below the rectangle labeled API) as well as profiling/tracing tools. The right

half of the figure depicts the application process with the original code of the program shown in the top part of the figure. The bottom two parts of the application are the snippets that are inserted into the program, and the runtime library that supports the Dyninst API. To perform our experiments, we installed Dyninst on a Sun Solaris-based platform.

III. Execution Model

A. Order of Events

Our execution model is depicted in Fig. 3. In this model, the application source code is augmented with "sensitivity lists." Each sensitivity list is attached to a module (a loop nest, subprogram, or function) and indicates the energy components that the module in question is sensitive to. `E-Optimizer` is a decision-making module which checks `E-Script` to determine the compilation strategy to choose, given the new energy constraint (e.g., remaining battery power). `E-Script` is a list that contains for each nest (or subprogram) the compilation strategies that should be activated based on energy constraints. After determining the optimization strategy, `E-Optimizer` asks `E-Compiler` whether there is already a compiled module in the "Compiled Code Repository," which corresponds to the new energy constraint. If there is, then `E-Compiler` supplies that module, which is subsequently inserted by `E-Optimizer` in the code and the execution resumes. If no such pre-compiled module exists in the repository, `E-Compiler` generates such a module and forwards it to `E-Optimizer`, which subsequently proceeds as explained above. It should be noted that, in our implementation, the variation in energy constraints is checked only when a sensitivity list check is being done (i.e., when a sensitivity list is reached during execution). This dynamic recompilation/linking-based execution environment has been implemented using Dyninst. In this implementation, `E-Optimizer` is a separate supervisory program that controls the code modifications to be performed on the target application.

We want to reiterate that our focus in this paper is on hiding as much compilation time as possible by overlapping it with application execution. Therefore, how the codes are optimized to adapt to changing energy constraints is beyond the scope of this paper (i.e., the approach proposed here is orthogonal to the set of compiler optimizations employed for reducing energy/execution cycles). It should be mentioned, however, that most of our optimizations applied during dynamic compilation target at memory banks, and increase/decrease the number of active banks to adapt to the energy constraints imposed. Our approach, however, can be made to work without any problem with any dynamic compilation framework and any set of compiler optimizations.

`E-Optimizer` comprises of two threads: `E-CtrlThr` (Controller Thread) and `E-CompThr` (Compilation Thread). `E-CtrlThr` monitors and controls the stopping/ starting and reconfiguration of the application. `E-CompThr` goes ahead of `E-CtrlThr`, predicts the next function/subroutine/nest that is energy sensitive, determines the appropriate compilation strategy for the new energy constraint, and compiles the new module and keeps it ready before the next sensitivity list for the energy sensitive region is reached. When the application reaches the next sensitivity list, it is stopped: If the dynamic compilation of the new module by `E-CompThr` is complete, it is inserted into the application and the application continues its execution. If the dynamic compilation is still in progress, the application waits until the compilation is complete before it can insert the new module and continue. In this framework, the only part that needs to be performed by the application programmer is to insert sensitivity lists at the appropriate places in the code. Our current implementation allows sensitivity lists to be added at the beginning of subprograms/functions and loops.

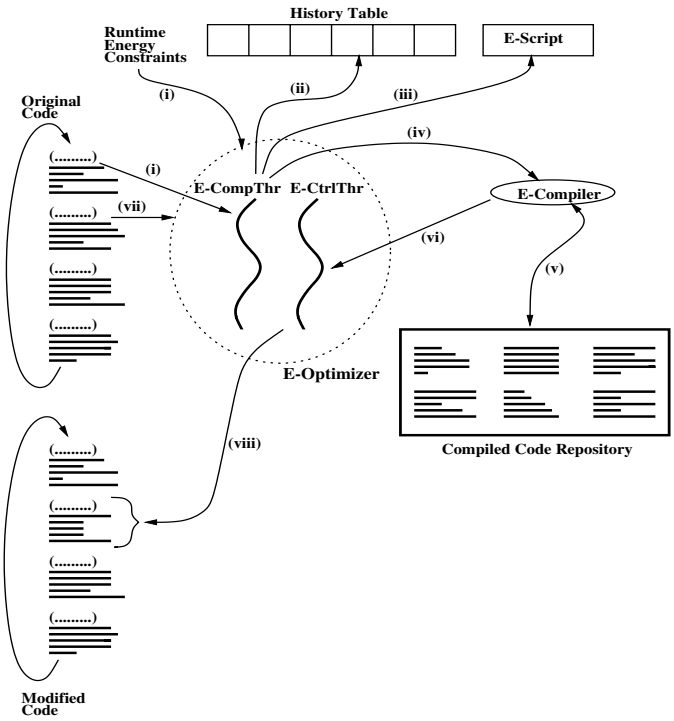

Fig. 3. Our execution model. The numbers attached to arrows indicate the order of events. This dynamic compilation framework has been implemented using Dyninst [2].

B. History-Based Next Module Prediction

The success of the proposed execution model hinges on the *prediction* of the next energy sensitive code region (module) to be executed in the application. On analyzing the function/subroutine/nest traces of several applications (that is, the order in which functions/subroutines/nests are executed), we observed regular patterns, which means that we can predict the next function/subroutine/nest to be executed with a reasonable accuracy. To benefit from this regularity, `E-Optimizer` maintains a history table (see Fig. 3). Every time the application program reaches an energy-sensitive region (annotated by a sensitivity list), it logs the function/subroutine/nest in the history table.[1] `E-CompThr` consults the history table before predicting the next energy sensitive region. Every time a sensitivity list is encountered, the compilation thread (`E-CompThr`) is activated to start the prediction and compilation of the next energy sensitive region. Savings in execution time are obtained when a correct prediction is made by `E-CompThr`. In the event of a misprediction, `E-Optimizer` invokes `E-Compiler` and compilation is done before the execution (incurring, of course, its performance overhead). Thus, the overall savings in compilation overhead are determined by the accuracy of the prediction and the history logged in the history

[1] *For array based applications, we focus on nests; that is, each nest is potentially an energy-sensitive module (code region). In contrast, for other applications, energy-sensitive modules are functions and subroutines. Unless stated otherwise, in our discussion, each nest (or function/subroutine) is considered as energy sensitive, and is augmented by a sensitivity list.*

TABLE I
BENCHMARK CHARACTERISTICS.

Benchmark	Source/Type	Size/Input File
btrix	Spec95	721KB
tomcatv	Spec95	836KB
vpenta	Spec95	770KB
hier	Motion Est.	310KB
full_search	Motion Est.	310KB
epic	MediaBench	test_image.pgm
rasta	MediaBench	ex5_c1.wav
181.mcf	Spec2000	inp.in

TABLE II
PREDICTION ACCURACIES.

Benchmark	Prediction Accuracy
btrix	99.95%
tomcatv	88.38%
vpenta	99.96%
hier	100.00%
full_search	100.00%
epic	91.02%
rasta	72.13%
181.mcf	87.52%

table.

Our current prediction strategy makes use of the history table as follows. If the current module to execute is `m1`, we search the history table to find when was the last time `m1` was called (invoked), and what was the method invoked following it. If this method (in the history list) is `m2`, we predict the next method to be called (after `m1`) as `m2`, and start pre-compiling it (if it needs to be recompiled). In other words, our approach assumes that if the most recent execution of `m1` has been followed by an execution of `m2`, the current execution of `m1` will also be followed by `m2`. Table II gives the prediction accuracies for the benchmarks used in this study. The prediction accuracies for array based codes are quite high due to the regular patterns in which loop nests are executed. That is, the loop nests in general are executed one after another with little (or no) control flow between them. For the other benchmarks, the prediction accuracy is dependent on the number of modules that are to be monitored and also on the regularity of their execution patterns. Overall, the prediction accuracies listed in Table II are very encouraging and imply that our pre-compilation based strategy can be successful in practice. However, we need to point out that a more sophisticated predictor can potentially generate even better predictions. In fact, we believe that it is even possible to parameterize the dynamic compiler to use different strategies at runtime based on a static profile.

IV. EXPERIMENTS AND RESULTS

A. Time Contribution of Dynamic Compilation

To evaluate our dynamic compilation based environment, we conducted experiments using eight benchmarks from different domains and benchmark suites. Table I gives important characteristics of these benchmark codes. The main reason for selecting these benchmarks is that we were able to run them through the Dyninst environment. The primary aim of our scheme is to hide the compilation time and thus reduce the overall execution time of applications in comparison to the default case, where dynamic compilation and execution are interleaved. Table III gives the best and the worst case (percentage) compilation times for each of the benchmarks in our suite when dynamic compilation and execution are fully interleaved (i.e., when no compilation time is hidden). To evaluate the worst case compilation time, each energy sensitive region is optimized and recompiled every time it is called before it can be executed. The compilation is based on the compilation strategies captured in the `E-Script`. In contrast, in the best case, each energy sensitive region is compiled only once during the course of entire execution, i.e., the first time the region is encountered. For every subsequent occurrence, the pre-compiled binaries from the Compiled Code Repository are used. Note that this best case represents the smallest amount of runtime that can be taken by the dynamic compiler. From the

TABLE III
THE BEST-CASE AND WORST-CASE TIMES FROM THE COMPILATION PERSPECTIVE (IN MILLISECONDS). THE VALUES WITHIN THE BRACKETS GIVE THE PERCENTAGE CONTRIBUTION TO THE TOTAL (COMPILATION PLUS EXECUTION) TIME.

Benchmark	Compilation Time (Worst Case)	Compilation Time (Best Case)	Execution Time (msec)
btrix	17651.69 [53.1%]	1810.51 [10.4%]	15599.96
tomcatv	22225.82 [80.3%]	1854.82 [25.4%]	5450.17
vpenta	20060.68 [36.5%]	1834.51 [5.0%]	34864.98
hier	3515.73 [36.7%]	1387.97 [18.7%]	6053.28
full_search	2468.43 [70.6%]	249.01 [19.4%]	1029.84
epic	90476.59 [82.0%]	1547.30 [7.2%]	19878.16
rasta	85515.07 [86.4%]	3138.66 [18.9%]	13436.02
181.mcf	5283755.26 [85.8%]	1490.7 [0.2%]	873999.61

results given in Table III, one can see that the compilation time constitutes more than 80% of the overall time in benchmarks tomcatv, epic, rasta and 181.mcf when the worst case scenario is considered. Even considering the best case scenario, it can be observed that, on the average, 13% of the total time goes to dynamic compilation. Consequently, an optimization strategy that hides this overhead (time) can be very useful in practice. In fact, the results in this table along with those in Table II provide a strong motivation for our research.

B. Reductions in Overall Execution Times

We recorded the total execution times with our strategy for various compilation probabilities (100%, 50%, and 25%) and compared them with the results when compilation and execution are interleaved. A 100% compilation probability implies that every module in the application has to be recompiled before it can be executed. On the other hand, a 50% (25%) compilation probability implies that only 50% (25%) of the modules in the application has to be recompiled before each time they need to be executed. Our experience with different applications indicate that most applications have compilation probabilities ranging from 25% to 80%. We performed experiments with these three different probabilities to cover a large spectrum. The experiments were executed on a Sun multi-processor machine running Solaris. The application process was bound to one of the processor (Processor 1) and the threads of `E-Optimizer` (i.e., `E-CtrlThr` and `E-CompThr`) were both bound to another processor (Processor 2). On analyzing the results obtained (see columns 2, 3, and 4 in Tables IV, V, and VI), one can make several observations. First, our approach reduces the time spent in dynamic compilation significantly. Specifically, with compilation probabilities of 100%, 50%, and 25%, the average reductions in the compilation time (the second column of the tables) are 32.06%, 20.45%, and 28.06%, respectively. However, to make a fair comparison, one needs to quantify the negative impact of our approach as well, that is, the increase in the execution time due to thread synchronization and contention. We can see from the third column in Tables IV, V, and VI that in some cases the increase in execution time is so much that it offsets all the benefits coming from parallelizing dynamic compilation (consider, for example, benchmarks such as rasta and 181.mcf with compilation probabilities of 50% and 25%). When one considers the total execution times (compilation time plus execution time) i.e., the fourth column in Tables IV, V, and VI, one sees an average of 10.35% and 4.07% reductions in total times with compilation probabilities 100% and 50%, respectively, and a 2.19% increase in the total time with a 25% compilation probability. Therefore, we can conclude that although we obtain

TABLE IV
PERCENTAGE IMPROVEMENTS DUE TO PARALLELIZING COMPILATION (WITH A 100% COMPILATION PROBABILITY).

Benchmark	2 Processors			3 Processors		
	Compl%	Excn%	Total%	Compl%	Exec%	Total%
btrix	23.11	-4.16	10.26	20.07	-0.24	10.50
tomcatv	30.80	-90.13	6.63	14.02	-5.32	10.15
vpenta	40.25	-2.38	13.17	38.69	-1.75	13.00
hier	63.75	-0.77	22.92	61.76	-0.68	22.25
full_search	27.06	-3.96	17.94	22.98	-2.28	15.55
epic	13.72	-39.36	4.01	28.66	-7.35	22.07
rasta	28.32	-162.50	2.00	36.71	-11.85	30.01
181.mcf	29.49	-129.02	5.93	38.25	-7.04	31.51

TABLE V
PERCENTAGE IMPROVEMENTS DUE TO PARALLELIZING COMPILATION (WITH A 50% COMPILATION PROBABILITY).

Benchmark	2 Processors			3 Processors		
	Compl%	Excn%	Total%	Compl%	Excn%	Total%
btrix	22.28	-0.67	8.34	15.99	0.38	5.65
tomcatv	8.48	-26.21	-2.25	9.92	-3.77	5.07
vpenta	33.19	-1.44	6.51	48.27	-1.09	6.64
hier	43.78	0.01	12.33	82.62	0.02	12.74
full_search	22.69	-2.72	12.17	33.17	-13.79	8.89
epic	11.63	-19.37	2.26	7.81	-4.51	3.69
rasta	12.50	-72.27	-6.42	16.03	-8.00	8.95
181.mcf	9.05	-28.50	-0.34	10.70	-9.65	4.84

some improvements, using only two processors does not bring impressive benefits.

These results led us to perform another set of experiments, where we used three processors. Specifically, the application process, as before, was bound to one processor (Processor 1), the controller thread (`E-CtrlThr`) was bound to a different processor (Processor 2), and the compilation thread (`E-CompThr`) was bound to a third processor (Processor 3). As can be seen from the columns 5, 6, and 7 of Tables IV, V, and VI, the results obtained using three processors are much better as compared to those obtained using two processors only. Specifically, with a compilation probability of 100%, the average reductions in the compilation time, execution time, and total time are 32.64%, -4.56%, and 19.38%, respectively. The corresponding values for compilation probabilities of 50% and 25% are 28.06%, -5.05%, and 7.05% and 14.91%, -1.59%, and 0.81%, respectively. These results indicate that our approach can make use of available processors effectively. It should also be emphasized that some of our applications, e.g., epic, rasta, and 181.mcf, are not amenable to chip-scale parallelism; so, we cannot use the available processors for reducing their execution times. However, it is still possible to use our approach to reduce their dynamic compilation times.

C. Impact of Compiling Critical Modules

Recall that in our experiments so far we considered all modules (functions/subroutines/ nests) as energy sensitive regions. In our next set of experiments, we analyzed the impact of dynamically compiling only a set of critical modules (instead of all modules). A critical module is the one that contributes to the overall execution time significantly. The experiments were performed assuming a 100% compilation probability. Table VII shows, for vpenta, the effect of compiling critical modules (nests) when execution and compilation are fully interleaved (i.e., when our scheme is not employed). The first column in this table lists the dynamically compiled critical nests in the application, the second column gives the compilation time, and the third column gives the overall time (compilation plus execution). It is easy to see that there is an optimum set of nests (Nest I + Nest II + Nest IV) such that, when compiled, lead to minimum overall time. In other words, for the minimum total time, it is important that only a specific set of critical modules need to be (predicted correctly and) dynamically compiled. In fact, as can be observed from this table, attempting to compile a larger set increases the total time. Similar results are presented in Table VIII when our approach is employed. Compiling only these critical modules (i.e., Nest I + Nest II + Nest IV) when execution and compilation are interleaved results in an improvement of 14.07% (for vpenta) in the total time. Overlapping execution and compilation yields an additional 13.98% improvement in the total time of the application, thereby producing a 28.05% overall performance improvement. Similar results have been observed with other benchmark codes as well, and those results are omitted here due to lack of space. One can conclude from these results that overlapping computation and compilation in addition with compiling only the critical modules yields the best results. In addition, we found that profiling can be of great help in determining the modules to compile. For example, in btrix, the seventh nest takes 95.2% of the time and is, therefore, a perfect candidate for compilation. Similarly, in 181.mcf, three subroutines, namely, refresh_potential, price_out_impl, and primal_bea_mpp take (together) 54.3% of the overall execution time. And, consequently, compiling only these critical modules generated between 20% and 30% improvements for these applications. As a result, a simple strategy for reducing the runtime overhead due to dynamic compilation would be (i) to profile the application to determine the critical modules and, (ii) to pre-compile these modules using the approach discussed in this paper.

TABLE VI
PERCENTAGE IMPROVEMENTS DUE TO PARALLELIZING COMPILATION (WITH A 25% COMPILATION PROBABILITY).

Benchmark	2 Processors			3 Processors		
	Compl%	Excn%	Total%	Compl%	Excn%	Total%
btrix	11.28	-0.51	2.52	4.45	0.22	1.26
tomcatv	14.54	-13.10	1.83	-6.12	-2.11	-4.49
vpenta	37.32	-0.47	5.89	63.42	-0.27	6.31
hier	37.27	0.43	9.22	60.86	0.32	9.27
full_search	-40.03	-4.85	-17.72	-9.55	-0.85	-4.40
epic	14.95	-11.19	2.87	8.87	-4.79	2.16
rasta	1.56	-49.92	-17.68	6.01	-6.32	1.19
181.mcf	-0.12	-10.11	-4.48	-8.59	1.08	-4.84

TABLE VII
VPENTA: IMPACT OF COMPILING ONLY CRITICAL LOOP NESTS.

Nests	Compile(ms)	Total(ms)
None	0	54375.86
Nest I	2874.97	55963.48
Nest I + Nest II	4824.32	58135.06
Nest I + Nest II + Nest IV	7319.83	46720.90
Nest I + Nest II + Nest IV + Next VIII	9692.83	59210.55
All Nests	20060.67	54989.12

D. Impact of Increasing the Number of Processors

In our experiments so far, we have used at most two processors for compilation purposes (in addition to the one that is executing the application itself). However, in some cases, allocating more processors to dynamic compilation can bring further reductions in overall execution time. Consider the example scenario depicted in Fig. 4. In this scenario, three processors are employed for performing dynamic compilation, whereas one processor is executing the application. Obviously, one might be able to get further benefits from even a larger number of processors depending on the application in question. It should be noted, however, that an opposite ar-

TABLE VIII
VPENTA: ABSOLUTE TIMES AND PERCENTAGE IMPROVEMENTS [DUE TO OVERLAPPED COMPUTATION AND COMPILATION OVER TABLE VII].

Nests	Compile (msec)	Total (msec)
Nest I	262.98 [90.85%]	52614.01 [5.99%]
Nest I + Nest II	472.14 [90.21%]	53585.28 [7.83%]
Nest I + Nest II + Nest IV	2652.98 [63.76%]	39124.29 [10.58%]
Nest I + Nest II + Nest IV + Next VIII	3169.78 [67.30%]	52648.76 [11.08%]
All Nests	11986.05[40.25%]	47747.07 [13.17%]

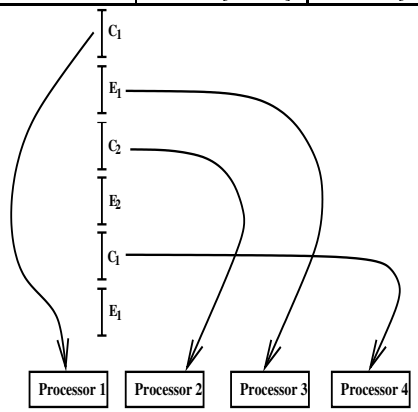

Fig. 4. An execution scenario where multiple processors are employed for dynamic compilation.

gument may defend using available processors for executing the application. As discussed earlier in the paper, such an approach may or may not be successful, depending on whether the application is amenable to module-level (loop nest-level or function/subroutine-level) parallelism. Specifically, if the application does not exhibit inherent parallelism, increasing the number of processors would not bring any benefits.

To study this tradeoff between parallelizing dynamic compilation and parallelizing application execution, we performed a set of experiments where we increased the number of processors and used them for compilation and execution. The results obtained are presented in Fig. 5 with a compilation probability of 50%. The top graph in Fig. 5 shows the results when the number of processors allocated for dynamic compilation is increased (and using only 1 processor for application execution). The bottom graph, on the other hand, gives the results when the number of processors allocated for application execution is increased (keeping the number of processors allocated for compilation at 2 except for the first bar where we have only 1 processor for compilation). We can make several observations from these two graphs. First, in general, employing the available processors for dynamic compilation (as opposed to application execution) generates better performance results. Specifically, the average performance improvements when using 5 processors are 9.49% and 7.31%, respectively, when the extra processors used for compilation and execution. Second, considering the top graph in Fig. 5, for each application, there is an optimum number of processors beyond which increasing the number of processors does not bring benefits. This is because of two main reasons. First, in some cases, accurately determining the multiple modules to pre-compile is not easy (that is, the prediction is difficult). Second, inter-thread communications can sometimes offset the benefits coming from employing multiple processors. Note that one can make a similar observation from the bottom graph in Fig. 5 as well. However, this time, the main reason for poor scalability is the inherent difficulty in parallelizing the application in question.

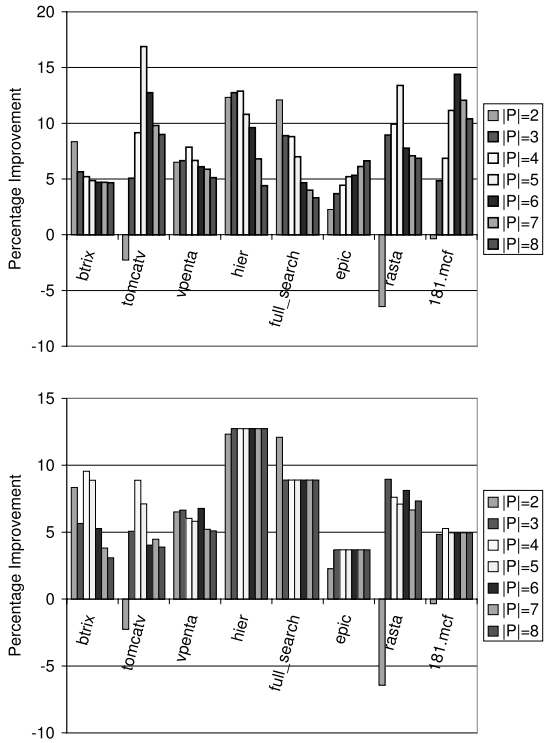

Fig. 5. Top: Percentage performance improvements with increasing number of processors employed for dynamic compilation. Bottom: Percentage performance improvements with increasing number of processors employed for application execution.

V. CONCLUDING REMARKS

The goal of this paper is to hide the time spent in dynamic compilation by overlapping it with application execution. We implemented a dynamic compilation/linking infrastructure that compiles/links program modules based on external energy constraints. Our strategy hides most of the dynamic compilation time by predicting the next module to be executed and by pre-compiling it. Our experimental results obtained using eight applications are very promising.

REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. *Technical Report HPL-1999-78,* HP Laboratories, 1999.

[2] B. R. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications, 14(4):317–329,* Winter 1994.

[3] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamically optimizing compiler for Java. In *Proc. the ACM Java Grande Conference,* June 1999.

[4] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proc. the ACM Conference on Programming Language Design and Implementation,* June 2000.

[5] K. Ebcioglu and E. R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. the International Symposium on Computer Architecture,* 1997.

[6] D. R. Engler. VCODE: a retargettable, extensible, very fast dynamic code generation system. In *Proc. the 23rd ACM Conference on Programming Language Design and Implementation,* 1996.

[7] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. Conference on Programming Language Design and Implementation,* May 1999.

[8] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software-Practice and Experience,* 31(8):717–738, 2001.

[9] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *Proc. SIGPLAN Conference on PLDI,* 1995.

[10] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer,* 28(11), 1995, pp. 37–46.

[11] P. Unnikrishnan, G. Chen, M. Kandemir, and D. R Mudgett. Dynamic compilation for energy adaptation. In *Proc. the International Conference on Computer Aided Design,* San Jose, CA, November, 2002.