# Using Dynamic Compilation for Continuing Execution Under Reduced Memory Availability

Ozcan Ozturk
Computer Engineering Department
Bilkent University
Bilkent, Ankara, Turkey
Email: ozturk@cs.bilkent.edu.tr

Mahmut Kandemir
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802
Email: kandemir@cse.psu.edu

***Abstract*—This paper explores the use of dynamic compilation for continuing execution even if one or more of the memory banks used by an application become temporarily unavailable (but their contents are preserved), that is, the number of memory banks available to the application varies at runtime. We implemented the proposed dynamic compilation approach using a code instrumentation system and performed experiments with 12 embedded benchmark codes. The results collected so far are very encouraging and indicate that, even when all the overheads incurred by dynamic compilation are included, the proposed approach still brings significant benefits over an alternate approach that suspends application execution when there is a reduction in memory bank availability and resumes later when all the banks are up and running.**

## I. Introduction and Motivation

Dynamic compilation has been identified as an effective means of adapting program execution to runtime conditions, and has been employed by past research for improving performance and reducing power consumption. In the former case, some program characteristics, which are not fully known at compile time, are revealed at runtime and the code is restructured (at runtime) to exploit these characteristics. For example, the values of some of the compile-time unknowns may become known at runtime and a dynamic compiler can restructure the program during execution to take advantage of these values. As another example, after some execution period, we may be able to identify the most frequently-executed functions/methods in the program and recompile them at runtime using a more powerful set of optimizations than the default ones. Most of the existing dynamic compilation platforms [1], [2], [9], [15], [21] are tuned to improve application performance at runtime. However, sometimes, an external condition (which has nothing to do with program variables or functions/methods) can force the program to be recompiled. An interesting example is from the domain of battery-operated embedded systems. When the remaining battery power during execution goes down below a certain level, it might be necessary (if possible) to recompile the program so that it consumes less power than the initial version. An example of such a dynamic compilation framework oriented to save power is presented [18]. Similarly, Wu et al [20] present a dynamic compilation framework for implementing DVS (dynamic voltage scaling).

As against these prior efforts, this paper presents and evaluates a dynamic compilation framework that is *resource availability oriented.* While one may think of many potential resource availability related scenarios where dynamic compilation can be useful, the specific scenario considered in this work deals with the case where one or more memory banks in a banked memory system become temporarily unavailable during application execution. When such a situation occurs, the proposed approach tries to continue execution using the remaining banks by restructuring the application code at runtime such that it uses only these operational banks. The amount of code we can continue to execute with the remaining available banks depends on a number of factors, including data dependencies across loop iterations, data-to-bank mapping, and data access pattern exhibited by the application being executed. When the unavailable banks later come back, we perform another dynamic compilation, this time to take advantage of these banks. It needs to be noted that the banks can become unavailable and later come back in any order, and each time a change takes place in bank availability, we consider dynamic compilation (but, whether dynamic compilation is actually invoked or not depends on other factors as well, as will be discussed later in the paper). Our focus is on array intensive embedded applications [4]. The particular scenario we target is one in which the banks do *not* lose data when they become unavailable. We target execution environments where an application manages the memory space allocated to it (i.e., no virtual memory or operating system support). That is, an application is responsible for managing its own memory space.

The important point to make is that our target scenario places emphasis on continuing execution in the existence of reduced bank (memory) availability. That is, our goal is to determine how far program execution can continue (after code restructuring) when some of the memory banks it normally uses become unavailable. We recognize that this approach is not appropriate for certain embedded systems where resources are controlled by a higher layer of software such as an operating system. However, our target environment here is a resource constrained one, which does not have any operating system or virtual memory support. We also understand that our method may not be the most preferable one for certain cases of embedded computing where one may simply want to stop application execution completely as soon as a bank becomes unavailable or where one may suspend execution until the unavailable banks come back. However, in many other cases, one may want to continue execution and make still some progress (especially when the unavailability lasts a while), even in the case where a number of banks are unavailable, i.e., when the amount of memory space dynamically changes. This is particularly important if we want to bring the system to a safe state gracefully, with as much progress in execution as possible, and the overheads involved in doing so are tolerable. These alternate options are depicted in Figure 1. Specifically, Figure 1(a) represents the execution with no memory bank problems (i.e., all banks are available all the time). In other three cases, it is assumed that (at least) one bank becomes unavailable at time $T1$ and comes back later at time $T3$. Figure 1(b) represents the case where execution is stopped as soon as a bank becomes unavailable. Alternatively, Figure 1(c) corresponds to the option where execution is suspended until all unavailable banks become available again. And finally, Figure 1(d) illustrates our approach where the execution continues even if some banks are not available. Notice that, the execution ends at times $T4$ and $T5$ with our scheme and the suspend-and-resume scheme, respectively, and in general we have $T4 < T5$.

We implemented the proposed dynamic compilation approach using a dynamic compilation framework built upon Dyninst [3], a dynamic code instrumentation tool, and performed experiments with 12 embedded benchmark codes. In our experiments, we simulated the behavior of the different execution scenarios where we change the frequency and pattern of bank unavailabilities and the frequency and pattern of their returns to the fully-operational state. The collected experimental results show that the proposed dynamic compilation based approach to runtime memory unavailability generates much better results than an alternate approach that suspends execution until all disabled banks come back, even if we account for all performance overheads brought by our dynamic compiler. Therefore, we believe that this paper is a step towards showing how dynamic compiler technology can be employed to cure some of the resource availability related problems faced by resource-constrained embedded execution platforms.

The next section discusses potential reasons for memory bank unavailability. Section III gives the details of our dynamic compilation framework. Section IV presents the results obtained using our implementation. Section V concludes the paper.

## II. Reasons for Memory Bank Unavailability

Bank availability may reduce during the course of execution due to several reasons. First, in a battery-operated embedded environment, one might want to reduce the number of banks when the battery level goes down beyond a threshold. This is because memory system is known to be a major energy consumer [4], and when available battery power goes down beyond a certain threshold, we may not be able to execute the program to completion by continuing to use all the banks. Instead, a better option in this case would be placing a certain number of banks into the low-power operating mode (where the contents of the bank are not destroyed but its power consumption is reduced significantly) and operate with the remaining banks until the battery power is re-charged to a certain level, at which point we start using a larger number of banks. The unavailability that occurs due to such power related reasons can last quite a long period of time. For example, using the dynamic compilation framework in [18], we performed experiments where we tried to adapt the execution of an application to dynamically changing battery constraints by turning off the select memory banks it uses. In these experiments, when the available battery power goes below a certain pre-set threshold, we shut down a memory bank[1] to save power. If the battery is not re-charged within some period of time, we shut down one more bank, and so on. When the battery starts to get re-charged, we bring the powered down banks back. We found that, an average unavailability duration for a given bank, can be very large, in some cases up to 28% of the total cycles of an application that completes in nearly 1.9 minutes.

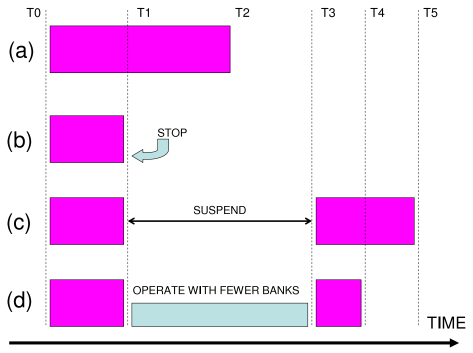

Fig. 1. Different execution scenarios. Application execution starts at time $T0$ and the original execution time is $T2 - T0$.

Another reason why the number of memory banks available to an application can be reduced at runtime is due to thermal issues [16], [10]. Specifically, when the chip temperature reaches a certain level (due to some other hardware components that are placed close by to the memory banks in the chip), it might be necessary to stop using some banks (but retain their data) until the temperature returns to normal. In this case, dynamic compilation can be an alternate option to pure circuit-based techniques. Yet a third reason is resource contentions in multi-process execution environments. For example, a very long DMA (direct memory access) operation can make certain banks unusable for the current application and it may be possible (and beneficial) to continue execution with the remaining banks.

It needs to be made clear that, while the actual reasons why some memory banks can be temporarily unavailable for a given application can change from one scenario/execution environment to another, our approach is quite general, and in fact, our dynamic compilation strategy is orthogonal to the underlying reasons for temporary bank unavailability. We would like to remind the reader that, in our target execution

[1]Shutting down a memory bank in this context means placing it into a low-power operating mode. In this mode, the memory bank still consumes some small energy (due to leakage current) but retains the data in it.

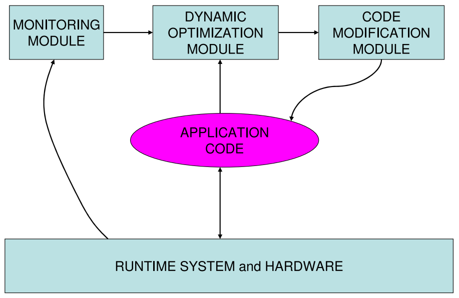

Fig. 2. High-level view of our dynamic compilation system.

environment, there is no virtual memory or operating system support, and therefore, the application and/or the compiler need to be able to cope with the bank availability related problems when they occur. Note also that, in all three unavailability scenarios mentioned above (i.e., power, temperature, and DMA related), the contents of the unavailable memory banks are retained.

## III. Details of Our Dynamic Compiler

Our goal in this section is to present the technical details of the dynamic compilation framework we developed. Figure 2 gives the high-level view of the proposed system. The Monitoring Module (MM) continuously monitors the memory banks to see whether any of the banks becomes unavailable and is explained in Section III-A. The Dynamic Optimization Module (DOM), detailed in Section III-B, determines the set of transformations that need to be applied to the code to ensure that the resulting code runs with the remaining (available) banks, i.e., it continues execution without using the temporarily unavailable banks. Since our focus is on array-dominated embedded applications, we employ Presburger arithmetic to formulate the problem. The Code Modification Module (CMM) modifies the code based on the transformations determined by DOM, and is described in Section III-C. For code modification purposes, we employ a polyhedral tool, called the Omega Library [6].

### A. Monitoring Module

The job of this module is to monitor the memory system and invoke dynamic compilation when there is a change in bank availability (either an increase or a decrease). In many scenarios, the job of MM is quite straightforward; it simply reads the bank availability information from a register that can only be updated by the runtime system. More specifically, when the runtime system decides to change the number of banks available to an application, it updates the contents of a special register called the *Bank Status Register* (or *BSR* for short), and this results in an interrupt and, within the interrupt service routine, we read the bank availability information from the BSR and invoke dynamic compilation. The overhead incurred by MM in this case includes those of going into the interrupt service routine, reading the contents of the BSR, and invoking dynamic compilation if necessary. As will be discussed shortly, in some cases, our approach may decide not to invoke the dynamic compiler even if there is a change in bank availability.

Another important point that needs to be clarified is regarding the detection of the point in execution where a change in bank availability occurs. As mentioned earlier, our focus is on array-dominated embedded applications. These applications are typically constructed using a series of loop nests, each operating on a subset of the arrays dynamically allocated in the application code. When a change in bank availability occurs at runtime, the dynamic compiler needs to know the current loop nest the execution is in and the loop iteration being executed. This is because the execution with the remaining banks should not repeat any computation that has already been performed. To do this, we keep track of the id (actually the address) of the current loop nest being executed in a reserved memory location. That is, each time a new loop nest is entered, we record its address into a reserved location and when the program is interrupted, the dynamic compiler quickly picks up the address from that location. Since the values of the loop iterators are kept in the register file, we learn the specific loop iteration during which the interrupt has occurred from the register file. In addition, we use a mechanism that allows us not to commit the writes to memory within a given loop iteration until that iteration finishes completely. This allows us to roll back and re-execute, when necessary, the iteration during which the bank unavailability is experienced.

### B. Dynamic Optimization Module

In this section, we discuss our mathematical framework for code restructuring to adapt the application to work with the reduced number of memory banks. We assume that array indices and loop bounds are affine functions of enclosing loop indices and loop-independent variables. We use Presburger arithmetic to capture and manipulate the loop executions and data accesses. Presburger arithmetic [5], [14] is the first-order theory of the natural numbers. There exist several tools for manipulating Presburger sets of affine constraints over integer variables. For our purposes, Presburger sets include affine expressions, logical and Boolean operators, and universal and existential quantifiers.

Let $\mathcal{I}_s$ be the iteration space of loop nest $s$, that is, the set of all iterations that will be executed by nest $s$. Similarly, we use $\mathcal{D}_p$ to denote the set of indices for array $p$ manipulated by the application. Assuming that the application has $S$ loop nests and $P$ arrays, we have $1 \leq s \leq S$ and $1 \leq p \leq P$. We use $\mathcal{R}_{s,p}$ to denote the set of references to array $p$ in loop nest $s$. Note that each element of $\vec{r} \in \mathcal{R}_{s,p}$ is a function that maps an iteration $\vec{I} \in \mathcal{I}_s$ to the data element indexed by $\vec{d} \in \mathcal{D}_p$. As an example, consider an array reference $U_p[i_1+1][i_2-1]$ (to a $n_{d_1} \times n_{d_2}$ array $U_p$) that appears within the body of a loop nest ($s$) with two loops ($i_1$ is the outer and $i_2$ is the inner loop), where $1 \leq i_1 \leq n_{i_1}$ and $1 \leq i_2 \leq n_{i_2}$. In this case, we have $\mathcal{I}_s$ = $\{(i_1, i_2) \quad | \quad (1 \leq i_1 \leq n_{i_1}) \quad \wedge \quad (1 \leq i_2 \leq n_{i_2})\}$; and $\mathcal{D}_p$ $\{(d_1, d_2) \quad | \quad (1 \leq d_1 \leq n_{d_1}) \quad \wedge \quad (1 \leq d_2 \leq n_{d_2})\}$; and $\vec{r}$ = $\{(i_1, i_2) \rightarrow (d_1, d_2) \quad | \quad (i_1, i_2) \in \mathcal{I}_s \quad \wedge \quad (d_1, d_2) \in \mathcal{D}_p \quad \wedge \quad (d_1 = i_1 + 1) \quad \wedge \quad (d_2 = i_2 - 1).\}$

We use $\mathcal{I}$ to denote the total set of loop iterations that will be executed by the application; i.e.,

$$\mathcal{I} = \bigcup_{s \in \mathcal{S}} \mathcal{I}_s,$$

where $\mathcal{S}$ is a set that holds the ids of all loop nests in the application. Let $f_p$ be a function that maps a given element of array $U_p$ to a bank in the memory; that is, $f_p : \mathcal{D}_p \rightarrow \mathcal{B}$, where $\mathcal{B}$ represents the bank space.[2] Selection of a suitable $f_p$ is beyond the scope of this paper; it can be made considering constraints such as bank locality or memory parallelism. We assume that $f_p$ is known to our dynamic compiler. We further define $\mathcal{E}_{p,b}$ as the set of elements of array $p$ mapped to bank $b \in \mathcal{B}$. In mathematical terms, we have:

$$\begin{aligned}\mathcal{E}_{p,b} \;=\; & \{\vec{d} \;\mid\; \exists \vec{I}, s, \vec{r} \;\text{ such that }\; \vec{I} \in \mathcal{I}_s \;\wedge \\ & \vec{r} \in \mathcal{R}_{s,p} \;\wedge\; \vec{r}(\vec{I}) = \vec{d} \;\wedge\; \vec{d} \in \mathcal{D}_p \;\wedge \\ & f_p(\vec{d}) = b\}.\end{aligned}$$

For convenience, we also define $\mathcal{E}_b$, the set of all data elements (coming, potentially, from multiple arrays) mapped to bank $b$, as:

$$\mathcal{E}_b = \{\vec{d} \;\mid\; \exists p \;\text{ such that }\; \vec{d} \in \mathcal{E}_{p,b}\}.$$

One of the important components of our dynamic compilation framework is its ability to determine the set of loop iterations that access only a given set of banks. This is important since when some of the banks cannot be used, the execution needs to continue with the remaining set of banks. But, in order to do that, we need to identify the set of loop iterations (from the set of iterations that need to be executed to finish program) that access the remaining banks so we can execute them to the extent allowed by data dependencies. We use $\mathcal{J}_b$ to denote the set of loop iterations that access *only* the array elements stored in bank $b$. We can express this set as follows. Let us first define an auxiliary set $\mathcal{J}'_b$ to capture the set of loop iterations that access the elements in bank $b$:

$$\begin{aligned}\mathcal{J}'_b \;=\; & \{\vec{I} \mid \exists s, p, \vec{d}, \vec{r} \;\text{ such that }\; \vec{I} \in \mathcal{I}_s \;\wedge\; \vec{r} \in \mathcal{R}_{s,p} \\ & \wedge\; \vec{r}(\vec{I}) = \vec{d} \;\wedge\; \vec{d} \in \mathcal{E}_b\}.\end{aligned}$$

Note that these iterations can access other banks as well. Now, we can write:[3]

$$\mathcal{J}_b = \{\vec{I} \mid \vec{I} \in \mathcal{J}'_b \;\wedge\; \neg b' \;\text{ such that }\; (b' \neq b \wedge \vec{I} \in \mathcal{J}'_{b'})\}.$$

What this set captures is that an iteration $\vec{I}$ belongs to set $\mathcal{J}_b$ if and only if it does not access another banks $b'$ which is different from $b$. If desired, we can extend this definition to multiple banks. For example, we can write a set $\mathcal{J}_{b_1,b_2}$ to represent the set of iterations that access array elements stored only in bank $b_1$ or bank $b_2$:

$$\begin{aligned}\mathcal{J}_{b_1,b_2} \;=\; & \{\vec{d} \mid (\vec{d} \in \mathcal{J}'_{b_1} \;\vee\; \vec{d} \in \mathcal{J}'_{b_2}) \;\wedge\; \neg b' \\ & \text{such that } \;(b_1 \neq b' \;\wedge\; b_2 \neq b' \;\wedge\; \vec{d} \in \mathcal{J}'_{b'})\}.\end{aligned}$$

It is easy to see that one can extend this approach to a larger set of banks as well. Based on our discussion above, suppose now that a memory system has $B$ banks ($b_1$, $b_2$, $b_3$, $\cdots$, $b_B$) and, at a particular point during execution, $L$ of these banks become temporarily unavailable, where $L < B$. Without loss of generality, we assume that these unavailable banks are $b_1$, $b_2$, $\cdots$, $b_L$. Therefore, our dynamic compiler should modify the application code such that it continues to operate using only the data stored in banks $b_{L+1}$ through $b_B$. To achieve such a code transformation, we first need to compute $\mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$, which can be done as explained above. This set gives us the loop iterations that access only banks $b_{L+1}$, $b_{L+2}$, $\cdots$, $b_B$, i.e., the iterations that do not access any of the unavailable banks.

### C. *Code Modification Module*

The job of this module is to generate the code that accesses only the elements in the set of available banks. To do this, we employ the Omega Library, a polyhedral tool that works with the Presburger sets. Omega Library is a set of C++ classes for manipulating integer tuple relations and sets [6]. It has been used in the past for different compilation tasks including dependence analysis, program transformations, generating code from transformations, and detecting redundant synchronization in parallel execution. In this work, we use the Omega Library to generate the loops that enumerate over the elements (iterations) in the Presburger sets such as $\mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$. Specifically, given a Presburger set $\mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$, the library's "codegen" utility generates the necessary code (typically loop nests) such that, when executed, iterates over the elements in set $\mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$. It needs to be noted however that we also need to generate code for the remaining iterations as well, as these iterations need also be executed once the unavailable banks become available again. For convenience we use the notation $\mathcal{K}_{b_{L+1},b_{L+2},\cdots,b_B}$ to capture the set $\mathcal{I} - \mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$. Overall, the new (restructured) code appears as being composed of two sets of program fragments; the first fragment contains the loops that enumerate iterations in $\mathcal{J}_{b_{L+1},b_{L+2},\cdots,b_B}$, whereas the second one involves the loops that enumerate iterations in $\mathcal{K}_{b_{L+1},b_{L+2},\cdots,b_B}$. It is important to emphasize here that this newly-generated code may need to be recompiled again if there is another update (in the course of execution) on the bank availability information. Another important issue regarding code generation is handling data dependencies in the application code being restructured.

## IV. Implementation, Setup, and Experiments

### A. *Implementation Details*

To implement our approach, we used the Dyninst tool [3]. Dyninst provides an Application Program Interface (API) that permits the insertion of code into a running program. Dyninst API is itself a C++ class library which can be included and directly called from a C++ program. This API is based on the technology developed as part of the Paradyn Parallel Performance Tools project [13] at the University of Wisconsin-Madison. A key feature of the Dyninst API interface is that it permits insertions and alterations in a running program, unlike other instrumentation tools such as EEL [8] or ATOM [17] which only allow code to be inserted into the binary before it starts to execute.

[2]We assume for simplicity that the bank space is linear, i.e., one dimensional. If it is multi-dimensional, we need to use $\vec{f_p}$ instead of $f_p$.

[3]$\neg x$ means "there is no x".

| | |
|---|---|
| L1 Size | 8KB |
| L1 Line Size | 32 bytes |
| L1 Associativity | 4-way |
| L1 Latency | 1 cycle |
| On-Chip Memory Size | 32MB |
| On-Chip Memory Access Latency | 16 cycles |
| Number of Banks for On-Chip Memory | 8 |
| Bus Arbitration Delay | 5 cycles |
| Replacement Policy | Strict LRU |

TABLE I
SIMULATION PARAMETERS.

| Benchmark Name | Brief Description | Source | Number of C Lines | Data Size (MB) | Execution Cycles (M) |
|---|---|---|---|---|---|
| Morph2 | Morphological operations and edge enh. | [7] | 878 | 24.7 | 2,314.4 |
| Disc | Speech/music discriminator | [7] | 2,022 | 18.4 | 1,877.3 |
| Jpeg | Lossy compression for still images | [7] | 771 | 15.3 | 1,705.1 |
| Viterbi | A graphical Viterbi decoder | [7] | 1,033 | 28.1 | 2,298.0 |
| Rasta | Speech recognition | [7] | 540 | 17.4 | 1,663.9 |
| 3Step-log | Logarithmic search motion estimation | [22] | 76 | 31.6 | 1,382.2 |
| Full-search | Full search motion estimation | [22] | 63 | 30.9 | 1,351.8 |
| Hier | Hierarchical motion estimation | [22] | 84 | 27.0 | 1,418.7 |
| Phods | Parallel hierarchical motion estimation | [22] | 114 | 13.8 | 1,338.9 |
| Epic | Image data compression | [11] | 3,530 | 17.6 | 1,672.1 |
| Lame | MP3 encoder | [12] | 18,612 | 24.9 | 2,592.8 |
| FFT | Fast Fourier transform | [12] | 469 | 20.2 | 1,996.2 |

TABLE II
BENCHMARK CODES USED IN OUR EXPERIMENTS.

## B. Setup

The simulation environment we use is built upon SIMICS [19], a system-level simulator. SIMICS can be used for building new virtual systems; high end architectural modeling; modeling of very large systems; and detailed timing models including interfacing to RTL models. The architectural model simulated is given in Table I. In this architecture, the chip contains a CPU, small L1 instruction and data caches, and an on-chip memory space divided into banks. Note that, unlike the case in high performance memory systems, memory banking employed in embedded systems normally does not use memory address interleaving. That is, each bank is assigned a set of consecutive memory addresses. In our experimental setup, we assume that the dynamic compiler is stored in a separate set of banks, which is not targeted by our approach.

To evaluate our approach, we used the benchmark codes listed in Table II. A common characteristic of these benchmarks is that they all are data-intensive applications that perform some sort of image, video, speech or network processing. The second column gives a brief description of each benchmark and the third column lists the source of each benchmark. The next column gives the number of C lines for each benchmark. The last two columns show, respectively, the total dataset size manipulated by the benchmarks and the total number of execution cycles in the ideal scenario where no bank unavailability occurs.

Clearly, our results are effected by the distribution of array elements across the available memory banks. Our default array distribution strategy is quite straightforward. We store the arrays one after another, starting with the first bank (no gap is allowed between two neighboring arrays). This storage tends to minimize the number of banks used. Later, we also discuss the results with two alternate array-to-bank mappings.

## C. Results

Our first set of results are given in Figure 3 and show the execution times of Morph2, Viterbi, Rasta, and Lame under the five different unavailability injection patterns (P1 through P5), *normalized* with respect to the execution time taken by an alternate approach (suspend-and-resume) that stops execution whenever any of the banks becomes unavailable and resumes execution when all the banks are up and running (see Figure 1(c)). That is, for each benchmark, the execution time under the suspend-and-resume scheme is set to 100%, and all the bars in Figure 3 are with respect to that. Each bar in Figure 3 is divided into two parts. The bottom part captures the time spent in actual execution (doing useful work), whereas the top part corresponds to the time spent within our dynamic compiler (i.e., the time spent in reading the bank availability information, determining the necessary code transformations, and transforming the code to adapt to the new bank availability). It can be seen from these results that our approach reduces the application execution times over the alternate scheme significantly. For example, the average reductions with the fault patterns P1 and P4 are 23.9% and 6.9%, respectively. It can also be observed that the overheads incurred by our dynamic compiler are more pronounced with P4 and P5 (as compared to the other three patterns), mainly because of the larger number of dynamic compilations incurred by these two patterns.

We next perform sensitivity studies with pattern P4. Recall that in P4 we make unavailable and, after some time, make available 4 of the 8 banks three times (each disabling is followed by an enabling). We study in Figure 4 the results when we change the number of times half of the banks become unavailable and later made available again. Each bar in these results represents the *average value* (taken over all 12 benchmarks) under the corresponding bank unavailability pattern. Note that 3+3 corresponds to the default P4 used so far in our experimental evaluation, meaning that banks $b_1$ through $b_4$ become unavailable and later become available three times during execution (at times $T/7$, $2T/7$, $3T/7$, $4T/7$, $5T/7$, $6T/7$, for a total application time of $T$). We see from these results that, as long as this number (i.e., the number of times a bank is made unavailable and later available again) does not exceed 4, the dynamic compilation based approach brings improvements over the suspend-and-resume based scheme. Since 5+5 exerts too much pressure on memory for such applications with short execution times, we believe that these results are promising. In addition to the results presented above, we also conducted experiments with the unavailability pattern P1 to investigate the impact of the length of the period between the banks becoming unavailable and later available. The results are similar across the different period lengths; however, we have better savings when the period in which the banks are unavailable is longer. If we take a closer look at the overheads incurred by our dynamic compiler, we see

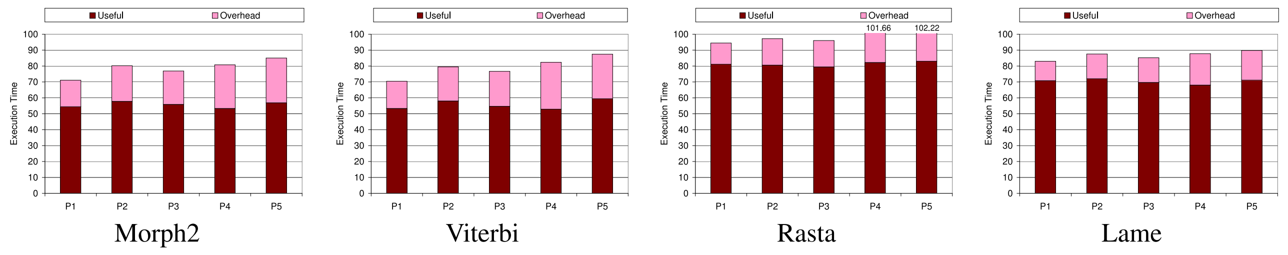

Fig. 3. Execution times of Morph2, Viterbi, Rasta, and Lame under five different bank availability patterns. Each bar is normalized with respect to a suspend-and-resume scheme, which stops execution whenever any of the banks are disabled and resumes execution when all the banks are up and running.

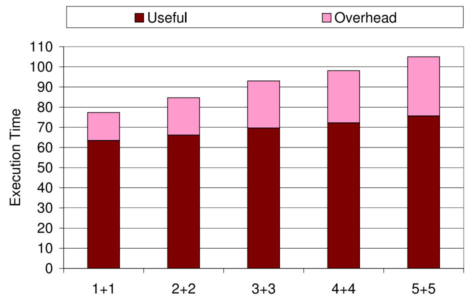

Fig. 4. Sensitivity analysis using availability pattern P4.

that, as expected, the dynamic optimization module, which determines the transformations to apply when the dynamic compiler is invoked, takes bulk of the overhead (80.6% on the average).

## V. Concluding Remarks

Dynamic changes in resource availability at runtime is an important problem to address in the embedded computing domain. The main contribution of this paper is to illustrate how a dynamic compiler can be used to adapt program execution at runtime to the modulations in memory bank availability. Focusing on array-dominated embedded applications and execution environments where an application manages the memory space allocated to it (without any operating system or virtual memory support), this paper defends a scheme that allows application execution to continue even if the number of banks available to the application is reduced. At the heart of our approach is a mathematical engine that isolates the set of loop iterations that accesses only a certain number of banks (i.e., available banks). We implemented our scheme on top of a dynamic code instrumentation tool, and performed experiments with 12 embedded benchmark codes. The experimental results collected so far show that the proposed dynamic compilation based approach generates much better results than an alternate approach that suspends execution until all disabled banks come back, even if account for all performance overheads brought by our dynamic compiler. To the best of our knowledge, this is the first study that uses dynamic compilation to address the memory availability problem at runtime.

## Acknowledgment

This research is supported in part by NSF Grants 0811687, 0720645, 0720749, and 0702519.

## References

[1] M. Arnold et al. Adaptive Optimization in the Jalapeno JVM. In *Proc. ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications,* Minneapolis, Minnesota, October 15-19, 2000.

[2] J. Auslander et al. Fast, Effective Dynamic Compilation. In *Proc. Symposium on Programming Language Design and Implementation,* May 1996.

[3] 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.

[4] F. Catthoor et al. *Data access and storage management for embedded programmable processors,* Kluwer Acad. Publ., Boston, 2002.

[5] J. Ferrante and C. W. Rackoff. The computational complexity of logical theories. Lecture Notes in Mathematics 718. Springer, 1979.

[6] W. Kelly et al. Code generation for multiple mappings. *Technical Report CS-TR-3317.1,* University of Maryland Institute for Advanced Computer Studies, December 1994.

[7] I. Kolcu. Personal communication.

[8] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *Proc. Symposium on Programming Language Design and Implementation,* 1995.

[9] S. Lee et al. Efficient Java exception handling in just-in-time compilation. In *Proc. Java Grande Symposium,* 2000, pp. 1-8.

[10] J. Li and J. F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proc. Symposium on High Performance Computer Architecture,* 2006.

[11] MediaBench. http://cares.icsl.ucla.edu/MediaBench/.

[12] MiBench. http://www.eecs.umich.edu/mibench/.

[13] B. P. Miller et al. The Paradyn parallel performance measurement tools. *IEEE Computer,* 28(11), 1995, pp. 37–46.

[14] C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *Proc. ACM Symposium on Theory of Computing,* 1978, pp.320-325.

[15] M. Serrano et al. Quasi-Static Compilation for Java. In *Proc. Symposium on Object-Oriented Programming, Systems, Languages, and Applications,* October 2000.

[16] K. Skadron et al. Temperature-aware microarchitecture: modeling and implementation. *ACM Transactions on Architecture and Code Optimization,* 1(1):94-125, Mar. 2004.

[17] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proc. Symposium on Programming Language Design and Implementation,* May 1994, Orlando, FL, pp. 196–205.

[18] P. Unnikrishnan et al. Dynamic compilation for energy adaptation. In *Proc. International Conference on Computer Aided Design,* 2002.

[19] Virtutech Simics. http://www.virtutech.se/

[20] Q. Wu et al. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proc. International Symposium on Microarchitecture,* 2005.

[21] B.-S. Yang et al. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proc. Symposium on Parallel Architectures and Compilation Techniques,* California, October 1999.

[22] N. D. Zervas et al. Code transformations for embedded multimedia applications: impact on power and performance. In *Proc. the ISCA Power-Driven Microarchitecture Workshop,* 1998.