Memory Energy Management Using Software and Hardware Directed Power Mode Control

V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin
Microsystems Design Lab
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
Technical Report: CSE-00-004

Abstract

The anticipated explosive growth of pervasive and mobile computing devices that are typically constrained by energy has brought hardware and software techniques for energy conservation into the spotlight. While there have been several studies and proposals for energy conservation for CPUs and peripherals, energy optimization techniques for selective operating mode control of DRAMs have not been fully explored. It has been shown that as much as 90% of overall system energy (excluding I/O) is consumed by the DRAM modules, serving as a good candidate for energy optimizations. Further, DRAM technology has also matured to provide several low energy operating modes (power modes), making it an opportunistic moment to conduct studies exploring the potential benefits of mode control techniques. This paper conducts an in-depth investigation of software and hardware techniques to avail of the DRAM mode control capabilities at a module granularity for energy savings.

Using a memory system architecture capturing five different energy modes and corresponding resynchronization times, this paper presents several novel compilation techniques to both cluster the data across memory banks as well as to detect module idleness and perform energy mode transitions. In addition, hardware-assisted approaches (called self-monitoring) based on predictions of module inter-access times are proposed. These techniques are extensively evaluated using a set of a dozen benchmarks. It is shown that we get an average of 61% savings in main memory energy using compiler-directed mode control. One of the self-monitored approaches gives as much as 89% savings (72% on the average), coming as close as 8.8% to the optimal energy savings that one can ever hope to get with memory module mode control. The optimization techniques are demonstrated to be invaluable for energy savings as memory technologies continue to evolve as well.

Keywords: Memory Architecture, Low Power, Low Power Compilation, Software-Directed Energy Management.
absr

1 Introduction

Computing devices for mobile and resource-constrained (embedded) environments are becoming the fastest growing market segment for the computer industry, even out-pacing corporate desktop, small office, and home computer sales. They are growing at a 20% annual rate and annual shipments are expected to grow to 30 million units by 2001. These environments demand components that are optimized for low cost, *low energy*, high performance, and small space. With energy taking the center-stage together with performance and packaging constraints, there has been a great deal of interest recently in examining optimizations for energy reduction from the hardware and software viewpoints.

From the hardware viewpoint, we find two complementary energy saving trends emerging. The first is the clustering of hardware components into smaller and less energy consuming components. An example is the multi-clustered architecture [47, 48, 18] where the register file, issue window, and functional units are distributed across multiple clusters on a chip. Zyuben and Kogge [48] show that such a multi-clustered architecture can be up to twice as energy efficient as wide-issue superscalar processors. The second trend is the support for different *operating modes* (*power modes/energy modes*), each consuming a different amount of energy. This provision is available in processors (e.g., the mobile Pentium III has five power management modes [31]), memory (e.g., the RDRAM technology [14] provides up to six power modes), disks [15, 28], and

other peripherals [38, 6, 7]. While these energy saving modes are extremely useful during idle periods, one has to pay a cost of *exit latency* (*resynchronization time*) for these hardware entities to transition back to the operational (active) state once the idle period is over.

From the software viewpoint, the research directions are on effective compiler, runtime, and application-directed techniques to selectively utilize as few hardware components as possible without paying performance penalties and transitioning the rest into an energy-conserving operating mode [43]. Industry is also recognizing the importance of supporting different energy modes and being able to transition between them on command, and is attempting to standardize the power management interface [3, 6].

While there have been several forays into hardware and software optimization techniques for energy savings in the context of processors [11, 43, 40, 41, 5, 10, 46, 19, 34], cache memories [39, 37, 25] and other peripherals [28, 16], such issues in the context of main memory (DRAMs) have mainly focussed on circuit and architectural techniques [24, 2] and data organizations [46, 12]. It has been observed [43, 26, 12, 46, 27] that memory system is a dominant consumer of the overall system energy, making this a ripe candidate for software and hardware optimizations, thus serving as a strong motivation for the research presented in this paper. This is especially true for mobile applications which are typically memory intensive (array dominant such as signal and video processing). In addition, applications are gradually becoming more data-centric with stringent memory requirements (both for storage and speed), causing vendors to incorporate large storage capacities into their offerings. Typically, a computer system contains several DRAM chips (organized in rows/banks and columns), with each of them consuming power even if it is not being currently used. It would be extremely valuable to explore techniques for selectively transitioning the unused memory modules into lower energy consumption modes (operating modes) whenever possible. Such techniques would not only be valuable in mobile and resource-constrained environments, but also for power management in normal desktop/server products due to cooling requirements. As with other hardware components, DRAM modules have started providing more mode control capabilities [32, 14], making the technology ripe for the ideas and techniques presented here.

With memory modules supporting multiple power modes and the ability to initiate a transition from one to the other, there are two main energy saving approaches for effecting such transitions that we explore. The first is the *compiler/software-directed approach*, where the application behavior is statically analyzed to detect idleness of memory modules for selective power down. This approach can be considered conservative since memory modules will not be transitioned to low power modes unless one is absolutely sure that a module will not be referenced for a while (at least for the time that it takes to bring it back to an operational state). However, its advantage is that there are *no* performance overheads due to resynchronization (exit latencies to active mode which consume not just time but also energy). At the other end of the spectrum is a hardware-assisted runtime approach (which we refer to as the *self-monitored approach* in this paper since the memory system automatically attempts to detect module idleness and transitions itself accordingly). This can adapt to (cycle-level) idleness that a compiler may not be able to detect, but there is a danger of incurring the resynchronization overheads due to mispredictions of the future idleness.

With the goal of minimizing energy consumption of memory by power mode control at memory bank granularity, this paper sets out to answer the following important questions:

- What hardware and enabling technologies are important for dynamically setting memory states? This may need to be explicitly CPU-directed (by application, compiler, or runtime system) or dynamically set based on memory reference behavior.
- What compiler-directed techniques can be developed to exploit memory reference behavior for dynamically turning off power? This depends on both how the data is allocated, and on being able to detect the distance between successive references and transition power modes accordingly without incurring any overheads.
- Given that transitions back to operational mode are expensive, how do we develop runtime self-monitored heuristics that can effect these transitions without incurring significant penalties? Their effectiveness will depend on how well we are able to predict inter-access times for different memory banks. Can we develop effective predictors that do not require too much real estate or power?
- What are the pros and cons of the above two approaches (compiler-directed and self-monitored), and when is one preferable over the other?

• Can we integrate the self-monitored and compiler-directed schemes to get the best of both worlds? How do these two techniques, together with the integrated approach, compare with the best (optimum energy consumption) that one can do for a given memory organization in terms of power and performance? This not only reveals how well we are doing, but can also give indications on the potential of future research.

• What is the impact of technology on the energy savings obtained by these techniques? Specifically, how do the number of energy modes, memory module configurations, and trends such as improved circuit techniques, newer technologies, and faster resynchronization times impact the energy savings obtained by our techniques?

This paper goes about investigating these issues in a systematic manner. First, the system architecture and software support for this research are explained. Next, a compilation framework for performing energy optimizations is presented. Using this framework, techniques are developed for grouping and allocating data structures in memory modules (called *clustering*), and for explicitly transitioning the memory modules to lower energy modes when not in use. In addition, self-monitored heuristics for automatic detection and prediction of idleness are proposed. Using twelve array-dominated benchmarks, these schemes are evaluated to identify their pros and cons. Array-dominated computations are typical in several image/media processing, virtual reality, signal processing, and scientific applications.

The rest of this paper is organized as follows. The next section explains the memory model for energy optimization. The experimental setup for the evaluations is given in Section 3. Section 4 presents the compilation techniques for energy optimization and the corresponding results. The self-monitored and integrated techniques are discussed and evaluated in Sections 5 and 6, respectively. Finally, Section 7 summarizes the contributions of this work and outlines directions for future research.

2 Memory Model for Energy Optimizations

2.1 Memory Architecture

Since the goal of this study is to explore the benefits of mode control at a DRAM module granularity (the smallest unit of energy management), we use a memory system that contains a number of modules organized into banks (rows) and columns as is shown pictorially in Figure 1 for a 4×4 memory module array. The proposed optimizations will, however, apply to most bank-organized DRAM memory systems. Accessing a word of data would require activating the corresponding bank and columns of the shown architecture. There are several ways of saving power in such an organization. We can either put the unused memory banks into a low power operating mode, or we could put the unused columns into a low power operating mode, or we could do a combination of the two. The savings with the latter two approaches (which can be beneficial when narrow-width data operands [10] are used) will depend largely on transfer unit sizes and the memory configuration. In this paper, we focus on the first approach only, and leave the other two for future research.

In addition, to keep the issue tractable, this paper bases the experimental results on a single program environment and does not consider the virtual memory system (i.e., we assume that the compiler directly deals with physical addresses). Exploring the influence of multi-programmed executions to study the impact of co-location of data structures *across* programs, and the presence of a virtual address translation are part of our future planned research. It should be noted that many embedded environments [21] operate without any virtual memory support, and the results from this paper would directly apply in those cases.

2.2 Operating Modes

We assume the existence of five operating modes for a memory module: *active, standby, napping, power-down,* and *disabled.*[1] Each mode is characterized by its *power consumption* and the time that it takes to transition back to the active mode (*resynchronization time*). Typically, lower the energy consumption, higher the resynchronization time [14, 32]. These modes are characterized by varying degrees of the module components being active. The major components of a DRAM module are the

[1] Current DRAMs [14] support up to six energy modes of operation with a few of them supporting only two modes. We collapse the read, write, and active without read or write modes into a single mode in our experimentation. However, one may choose to vary the number of modes based on the target DRAM.

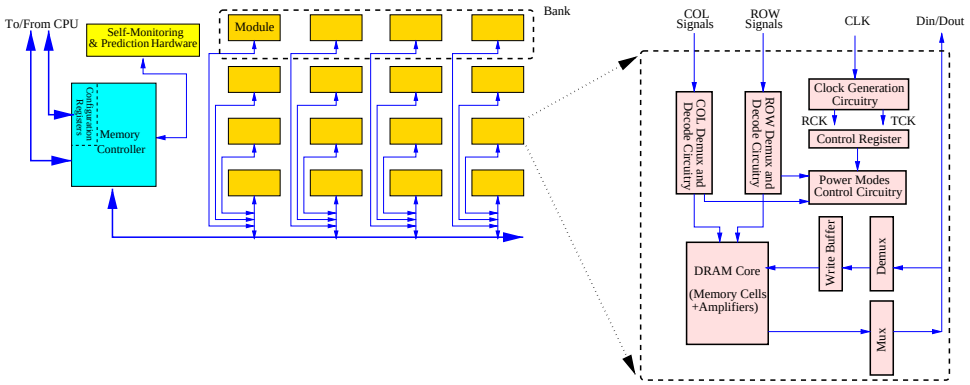

Figure 1: Memory system architecture.

clock generation circuitry, ROW (row address/control) decode circuitry and COL (column address/control) decode circuitry, control registers and power mode control circuitry, together with the DRAM core consisting of the precharge logic, memory cells, and sense amplifiers (see Figure 1). The clock generation circuitry is used to generate two internal clock signals (TCK and RCK) that are synchronous with an external system clock (CLK) for transmitting read data and receiving write data/control signals. The packets received from the ROW and COL signals can also be used to switch the power mode of the DRAM. The details of the power modes are discussed below:

• *Active:* In this mode, the DRAM module is ready for receiving the ROW and COL packets and can transition immediately to read or write mode. In order to receive these packets, both the ROW and COL demux receivers have to be active. As the memory unit is ready to service any read or write request, the resynchronization time for this mode is the least (zero units), and the energy consumption is the highest.

• *Standby:* In this mode, the COL multiplexers are disabled resulting in significant reduction in energy consumption compared to the active mode. The resynchronization time for this mode is typically one or two memory cycles. Some state-of-the-art RDRAM memories already exploit this mode by automatically transitioning into the standby mode at the end of a memory transaction [14].

• *Napping:* The ROW demux circuitry is turned off in this mode, leading to further energy savings over the standby mode. When napping, the DRAM module energy consumption is mainly due to the refresh circuitry and clock synchronization that is initiated periodically to synchronize the internal clock signals with the system clock. This mode can typically consume two orders of magnitude less energy than the active mode, with the resynchronization time being higher by an order of magnitude than the standby mode.

• *Power-Down:* This mode shuts off the periodic clock synchronization circuitry resulting in another order of magnitude saving in energy. The resynchronization time is also significantly higher (typically thousands of cycles).

• *Disabled:* If the content of a module is no longer needed, it is possible to completely disable it (saving even refresh energy). There is no energy consumption in this mode, but the data is lost. While one could envision transitioning out of disabled mode by re-loading the data from an alternate location (perhaps another module or disk) and/or just performing write operations to such modules, we do not consider such cases in this paper.

When a module in standby, napping, or power-down mode is requested to perform a memory transaction, it first goes to the active mode and then performs the requested transaction. Figure 2 shows possible transitions between modes (the dynamic energy consumed in a cycle is given for each node) in our model.[2] The resynchronization times in cycles (based on a cycle time of 2.5ns) are shown along the arrows (we assume a negligible cost ϵ for transitioning to a *lower power* mode). The model is flexible enough to take in different values for energy consumption and resynchronization costs, and the default values used are the ones given in Figure 2. While one could employ all possible transitions given in this figure (and maybe more), our compiler-directed approach only utilizes the transitions shown by solid arrows. The self-monitored approaches, on the other hand, can exploit two additional transitions: from *standby* to *napping*, and from *napping* to *power-down*. The energy values

[2] We do not consider leakage power in this paper.

shown in this figure have been obtained from the measured current values associated with memory modules documented in memory data sheets (for a 3.3V, 2.5ns cycle time, 8MB module) [14]. The resynchronization times are also obtained from data sheets. These values define our *base configuration* and Section 4.5 investigates the impact of varying some of these parameters.

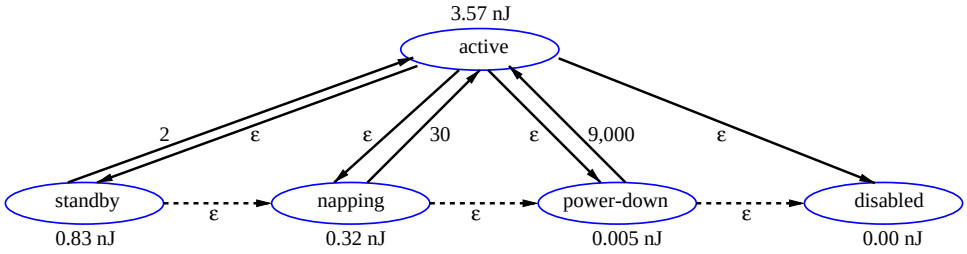

Figure 2: Power modes utilized.

2.3 System Support for Power Mode Setting

Typically, several of the DRAM modules (that are shown in Figure 1) are controlled by a memory controller which interfaces with the memory bus. The interface is not only for latching the data and addresses, but is also used to control the configuration and operation of the individual modules as well as their operating modes. For example, the operating mode setting could be done by programming a specific control register in each memory module (as in RDRAM [14]). Next is the issue of how the memory controller can be told to transition the operating modes of the individual modules. This is explored in two ways in this paper: *self-monitored* and *software-directed*.

In the self-monitored approach, there is a Self-Monitoring and Prediction Hardware block (as shown in Figure 1) which monitors ongoing memory transactions. It contains some prediction hardware to estimate the time until the next access to a memory bank and circuitry to ask the memory controller to initiate mode transitions.[3] The specific hardware depends on the prediction mechanism that is employed and will be discussed later in the paper.

In the software-directed approach, the memory controller is explicitly told to issue the control packets for a specific module's mode transitions. We assume the availability of a set of configuration registers in the memory controller (see Figure 1) that are mapped into the address space of the CPU (similar to the registers in the memory controller in [22]). Programming these registers using one or more CPU instructions (stores) would result in the desired power mode setting. This brings up the issue of which CPU activity needs to be able to issue such instructions. The memory control registers could potentially be mapped into the user address space directly, making it possible for the application/compiler to directly initiate the transitions. However, there are a couple of drawbacks with this approach. The first being that powering down modules which are shared with other applications brings up the protection issue. The other problem could be that one program does not have much knowledge of the memory activity of other programs, and will thus not be able to accommodate more global optimizations. With two or more applications sharing a memory module, the operating system may be a better judge of determining the operating (power) modes. So, the other option is to make the issuance of these instructions a privilege of the operating system, with the compiler/application availing of this service via a system call. Since the focus of this paper is to explore the potential benefits of memory module energy optimizations, we focus on a single program environment and assume that the registers are directly mapped into user space (so, they can be controlled by the compiler).

Regardless of whether a power mode transition is initiated by a self-monitored or software-directed mechanism, a graceful recovery to the operational mode is needed to service a read/write operation. This can create a problem because most current memory buses are synchronous, making it necessary for the operation to be complete within a specified number of bus cycles. However, transitions back to operational modes (active) can be expensive. As a result, the read/write operation can result in bus errors, making it necessary for the operating system to handle them appropriately. The exception handler can examine

[3] Limited amount of such self-monitored power-down is already present in current memory controllers (e.g., Intel 82443BX [22] and Intel 820 Chip Set [23]). However, the number of power modes and prediction hardware that we explore here are significantly more sophisticated.

status information in the memory controller to find out what state the referenced module is currently in, and can appropriately idle and re-issue the operation, or can use some latency tolerance techniques. In fact, the compiler-directed strategy discussed later in this paper uses the latter approach by issuing power up (to active) transitions *ahead of the use* of the corresponding modules. This is analogous to prefetching [30, 36] to hide memory latencies, and we can incorporate many of those ideas here as well.

3 Experimental Setup

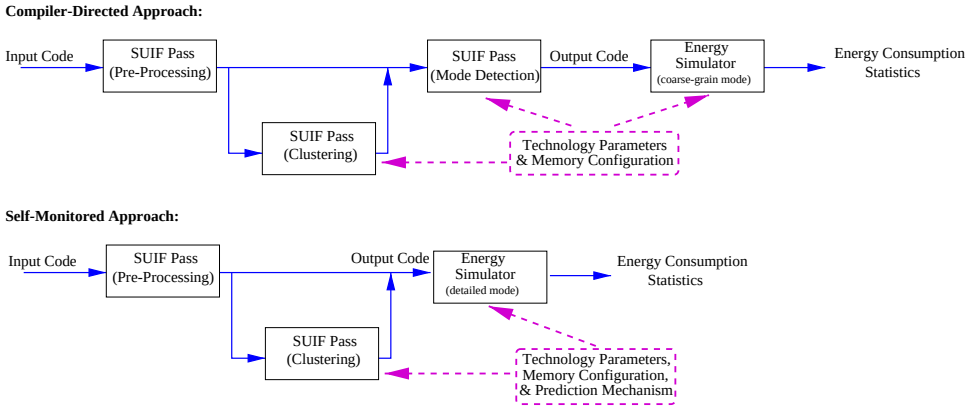

Figure 3: Experimental setup.

The compiler-directed approach presented in this paper has been implemented within the SUIF compilation framework [4]. Specifically, we have implemented two complementary techniques within SUIF. The first technique analyzes the input code and determines the points where operating mode instructions should be inserted (shown as Mode Detection in Figure 3). As will be explained in Section 4.4, it also applies necessary loop transformations [45] to make explicit the program points where these mode instructions are to be inserted. The second technique implements clustering, which basically places the data structures with *similar life patterns* into the same memory modules whenever possible (shown by a box marked Clustering in Figure 3). Clustering is done by *modifying the order of array declarations* and by inserting necessary paddings [33] as needed (see Section 4.2). Both techniques also use a common Pre-Processing pass which analyzes the input code and converts it to a version with as many independent loop nests as possible. This is done using a technique similar to that proposed by McKinley et al. [29]. Each independent nested loop is called a *phase* in this paper. In the compiler-directed approach, this is the smallest program unit for which we determine a power management strategy using different operating modes. The cycle estimates for the nests were obtained from actual executions of the programs on an UltraSparc5 architecture (operating at 360 MHz with Solaris 2.7) and these estimates were used for *all* our simulations. After the mode detection pass, the energy consumed is determined by an Energy Simulator based on the number of cycles spent in each of the power modes using the technology and memory configuration parameters.

In the self-monitored approach, the code after pre-processing can either be clustered or not, before it goes to Energy Simulator. The simulator computes the energy using cycle-by-cycle simulation of the memory accesses for the entire program execution. Note that in the compiler-directed approach, the simulator uses a coarser level of simulation (*phase granularity*), while the self-monitored approach does a more detailed (*cycle granularity*) simulation. For the self-monitored approach, the simulator took up to 3 hours (per simulation run) as compared to less than a minute for the compiler-directed approach.

Figure 4 gives the salient characteristics of the twelve benchmarks used in this paper. Our suite contains three image processing programs (*full_search* (6), *matvec* (7), and *phods* (9)) and nine codes manipulating large multi-dimensional arrays. The fourth column of the figure shows the total input sizes in megabytes. The fifth column gives the memory energy consumption (in milliJoules) when no energy optimization is applied. The sixth column gives the time (in seconds) spent in the implemented SUIF passes (i.e., pre-processing, clustering, and mode detection) in the compiler-directed mode (on a 360 MHz

Benchmark Number	Benchmark Name	Source	Data Size (MB)	Base Energy (mJ)	Compile Time (sec)
1	*adi*	Livermore	48.0	3.38	0.053
2	*dtdtz*	Perfect Club	61.8	2.55	0.046
3	*bmcm*	Perfect Club	39.9	3.93	0.049
4	*btrix*	Spec'92	47.7	2.49	0.193
5	*eflux*	Perfect Club	33.6	413.23	0.099
6	*full_search*	[9]	33.0	337.75	0.120
7	*matvec*	[8]	16.0	675.75	0.054
8	*mxm*	Spec'92	48.0	10.70	0.029
9	*phods*	[13]	33.0	1,586.25	0.122
10	*tomcatv*	Spec'95	56.0	119.80	0.093
11	*vpenta*	Spec'92	44.0	506.68	0.130
12	*amhmtm*	Perfect Club	48.1	7.40	0.054

Figure 4: Benchmark codes used in the experiments and their important characteristics.

UltraSparc5 workstation). As can be seen, the compilation overheads for implementing our scheme are negligible. Unless explicitly stated otherwise, a 8×1 memory module array (i.e., 8 banks with 1 module per bank) with 8 MB modules is used in our experiments. Consequently, module granularity and bank granularity of mode control achieve the same purpose. Thus, in the remainder of the paper, we use the words *module* and *bank* interchangeably.

It should be noted that caches can also have a significant impact on the energy savings of the proposed techniques. However, we conservatively limit our experimental strategy so that all memory accesses go to the main memory. We anticipate even better potential for memory power mode exploitation with the fewer memory accesses that would be expected in the presence of a cache. We report the influence of the cache using one set of preliminary experiments later in the conclusions (Section 7).

4 Compiler-Directed Energy Management

In this section, we present our compiler-directed approach to energy management for DRAMs. We start with compiler-directed operating mode selection and then present our data clustering technique. Finally, we present experimental numbers.

4.1 Operating Mode Management

The goal of our compiler-directed mechanism is to detect idle periods (inter-access times) for each memory module, and to transition it into a lower power mode *without paying any resynchronization costs*. Consequently, if the inter-access time is T, and the resynchronization time is T_p (assuming less than T), then the compiler would transition the module into a lower energy mode (with a unit time energy of E_p) for the initial $T - T_p$ period (which would consume a total $[T - T_p]E_p$ energy), activate the module to bring it back to the active mode at the end of this period following which the module will resynchronize before it is accessed again (consuming $T_p E_a$ energy during transition assuming that E_a is the unit time energy for active mode as well as during the transition period). As a result, the total energy consumption with this transitioning would be $[T - T_p]E_p + T_p E_a$ without any resynchronization overheads, while the consumption would have been $T E_a$ if there had been no transitioning (this calculation considers only the idle period). This is provably the minimum amount of energy since any other point of transitioning to a lower energy mode or resynchronizing would incur a higher overall energy consumption and/or resynchronization cost. Given that we have a menu of low energy modes to transition into, the compiler can evaluate all possible choices (low power modes) based on the mode energy, corresponding resynchronization times, and inter access time, to select the best choice. Note that the compiler can select different low power modes for different idle periods of the same module depending on the duration of each idle period. When the inter access time is ∞ (i.e., there is *no* next access), the module can be put into disabled mode.

4.2 Compiler-Directed Clustering

Our objective in clustering is to group the related (similar lifetime access patterns) array variables together so that they can be placed in the same memory modules. This increases the likelihood of transitioning a memory module to a lower energy mode. On the other hand, placing variables that are accessed at different points of the execution in the same module would result in its longer residence in the active mode.

We assume that the default allocation of variables is in program declared order. Since the compiler is directly working with physical addresses, it is relatively straightforward to determine the memory modules that different statically declared variables reside in. It should be noted that (depending on size of the banks and arrays) a single array variable can occupy multiple banks, and similarly, a single bank may hold multiple array variables.

Declaration order of array variables may have nothing to do with their access profiles and life times. Consequently, this order rarely leads to opportunities for effective use of low power operating modes. Our strategy is to analyze the program and determine the arrays with similar access behavior and use this information to *modify the declaration order of array variables* so that those with similar behavior are declared consecutively (and hopefully will map in the same modules as arrays are allocated in declaration order). Note that this approach requires minimum modifications to the source code. The disadvantage is that depending on the array and bank sizes, the resulting module assignments may not necessarily be energy efficient, especially if the arrays are smaller and some banks contain a large number of (and possibly unrelated) array variables, or some large arrays are divided across several banks. To eliminate this effect, we implement a modified version of this approach, which attempts to perform *bank alignment of arrays* as long as doing so does not increase the total number of required banks. Note that even this improved version is not as aggressive as one might expect. Theoretically, the best results would be obtained if the compiler is given complete control over memory allocation. That is, instead of just ordering the declaration sequence and performing alignment, it would force a specific data set (or even a portion of it) to be placed in a given part of a given bank. While we think that this could potentially achieve better energy savings (despite its complexity for both the compiler as well as for addressing), this is not considered here.

Our compiler algorithm reorders the declaration of array variables (i.e., clusters them) in six steps. The first step is a program analysis that keeps for each array variable a record of its name, size (in bytes), and life time. At the end of this step, we obtain an *array access profile* information that is shown in Figure 5 for *vpenta*, a floating-point code from the Specfp benchmark suite (arrays are labeled from `U1` to `U8`). Each phase corresponds to a nested loop and a $\times$ indicates that the array is accessed in the corresponding phase.

Phase Number	Array Variables							
	U1	U2	U3	U4	U5	U6	U7	U8
1	×			×	×	×	×	
2		×		×	×		×	×
3		×		×	×		×	×
4	×	×	×	×	×	×	×	×
5	×	×	×			×	×	×
6	×	×	×			×	×	×
7		×	×					×
8		×	×					×

Figure 5: Array access profile for *vpenta*.

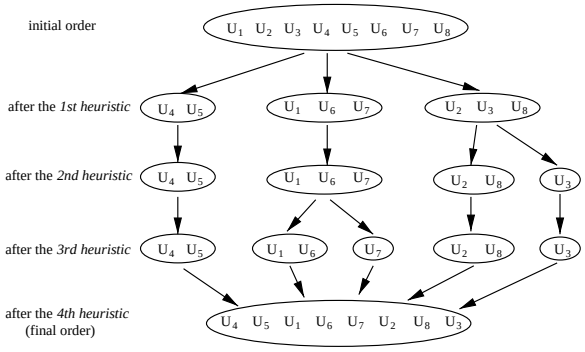

Figure 6: Applying our heuristics to *vpenta*.

Subsequently, the compiler goes through a sequence of four heuristics (steps 2 through 5) that divide the array variables into groups. Each heuristic respects the grouping imposed by a previous heuristic.

• *1st heuristic*— Array variables with the same *last usage phase* (LUP) are placed in the same group. The rationale is that if two array variables have the same LUP, they *both* can be assumed to be dead after that phase (and the corresponding memory module holding them can be disabled if there are no other live array variables in that module).

• *2nd heuristic*— Within each LUP group, the array variables are divided into subgroups based on their *first usage phase* (FUP). This helps keep the bank holding the array variables with the same FUPs in a low power mode until it needs to be first

accessed.

• *3rd heuristic*— Array variables within the subgroup from the previous heuristic are divided further into subgroups based on the $\times$ pattern of the corresponding columns in the array access profile. If two or more columns have the same $\times$ pattern (i.e., if they have $\times$ in the same phases), they are kept in the same subgroup. This helps identify the closely related variables for co-location on memory banks.

• *4th heuristic*— This heuristic is used to reorder array variables within the subgroups from the previous heuristic so that the array at the boundary of one group has a relatively close '$\times$ pattern' with the array at the boundary of its neighboring group. For example, if array variables {`V1`, `V2`, ...} and array variables {`W1`, `W2`, ...} are neighboring groups from the 3rd heuristic (with the former to the left of the latter), we attempt to place `Vi` as the rightmost array in the former group and `Wj` as the leftmost array in the latter group if they have the most similar $\times$ pattern (after comparing all possible combinations).

Figure 6 shows the groupings after using the four heuristics for the example array access profile shown in Figure 5. (In this example, the fourth heuristic does not have any effect.)

Finally, as a last step, we make a pass over the new declaration order and (taking into account the number and size of banks) try to modify the bank assignment (not the declaration order) so that the large (multi-bank) arrays are assigned into dedicated banks. This is currently done by array padding [33].

Our current implementation places scalar variables in memory modules after array variables have been placed. Whenever possible, it places a scalar variable in a memory bank if that scalar does not modify the bank access profile (explained next) of that bank.

4.3 Determining Bank Access Profiles and Modes

In order to perform mode control, it is necessary for the compiler to find bank access times. This requires translating *array access profiles* to *bank access profiles* (an example of which is shown in Figure 7) taking into account the memory configuration (number of banks, modules per bank, and module size). A $\times$ in the bank access profile indicates that the corresponding bank needs to be active during the execution of the corresponding phase. In a given bank profile, the entries without $\times$ represent opportunities for energy optimization. As an example, the left table in Figure 7 gives the bank access profile corresponding to the array access profile in Figure 5, with a declaration order `U1`, `U2`, `U3`, `U4`, `U5`, `U6`, `U7`, `U8`, and assuming that all the arrays are of equal size except `U3` (which is four times larger than others) and we have 8 banks, each capable of holding two of the arrays other than `U3`. In other words, the bank assignment is [`U1`,`U2`], [`U3`,`U3`], [`U3`,`U3`], [`U4`,`U5`],[`U6`,`U7`], [`U8`,$\emptyset$], [$\emptyset$], where each [ ] corresponds to a bank that contains (portions) of arrays ($\emptyset$ denotes empty space). Note that 35 out of 64 entries are active whereas the rest corresponds to idle state (i.e., 29 idle states). In general, clustering attempts to increase the number of idle states in the bank access profile. The bank access profile of the clustered version of this example is given in the table on the right side of Figure 7. We note that this has 11% better (more idle states) than the one on the left. If we have only 4 banks (each capable of holding four arrays expect `U3`) instead of 8, the optimized order and alignment determined by our approach results in a 21% improvement.

After determining the bank access profile and detecting the idle slots (states), for each bank we can determine suitable operating modes. Note that the modes can be determined for each bank independently using the energy consumption, resynchronization times and inter-access times by the approach explained earlier in Section 4.1. Essentially, the free slots in the profile are transitioned to an appropriate lower energy mode.

4.4 Automatic Insertion of Mode Instructions

The last part of the compilation is to insert suitable (operating) mode transitioning instructions in the program code. During processing of the source code, we do not actually insert any instruction, but just place markers (as placeholders). Later, during low-level optimization, we insert the actual mode transition instructions.

The time for issuing the mode transitions is very important for energy saving and performance. If they are issued too early, they will cause unnecessary power consumption (by putting the module into the active mode long before needed). On the other hand, if they are issued too late, the module may not be in the active mode when it is needed, leading to a performance loss. Since most of our optimizations are on array-based applications, it may be reasonable to choose the number of loop iterations

Phase Number	Bank Number							
	B0	B1	B2	B3	B4	B5	B6	B7
1	×			×	×			
2	×			×	×	×		
3	×			×	×	×		
4	×	×	×	×	×	×		
5	×	×	×		×	×		
6	×	×	×		×	×		
7	×	×	×			×		
8	×	×	×			×		

Phase Number	Bank Number							
	B0	B1	B2	B3	B4	B5	B6	B7
1	×	×						
2	×		×	×				
3	×		×	×				
4	×	×	×	×	×	×		
5		×	×		×	×		
6		×	×		×	×		
7			×	×	×	×		
8			×	×	×	×		

Figure 7: Example bank access profiles. Left: original, Right: optimized.

as the basic unit for measuring time (i.e., the mode transition instructions will be issued at the boundaries of iterations), requiring that all times be converted to iteration counts.

To avoid resynchronization costs, the power up instructions for a given phase (nested loop) need to be executed in the *previous* phase. The number of iterations before the termination of the previous phase when the mode transition instruction needs to be executed can be calculated as

$$\left\lceil \frac{T_r}{T_{sp}} \right\rceil + T_{ov}$$

where T_r is the resynchronization time in cycles, T_{sp} is the length of the *shortest path* (in terms of cycles, see Section 3) through the loop body (of the previous phase), and T_{ov} is the software overhead (in terms of loop iterations) introduced by inserting a mode control instruction in the code. Since finding the optimal time for issue is very difficult, especially with superscalar architectures, we opted to use the shortest path [44] conservatively. Though T_{ov} may be critical for some loop nests with very small loop bodies, we did not notice any significant impact of it in our experiments.

Once the issue point for the mode transition instruction is determined, our approach uses *loop splitting* [45] to make this point more explicit. As an example, consider the example in Figure 8(a) that has two nests with the second nest demanding active mode for a specific module which is in a low power mode during the first nest. Assuming that the resynchronization time is 60 cycles, the shortest path through the body of the first nest is 5 cycles, and the software overhead is 3 iterations, the mode instruction should be issued 15 (= 60/5+3) iterations ahead. Assuming that $M \geq 15$, our compiler converts the program fragment above into that shown in Figure 8(b). Note that the placeholder is 15 iterations before the end of the first nest.

```
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
    {U[i][j]+=V[j][i]+1;
    V[i][j]=V[j][i]-k*U[i][j];}
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
    { ... ;}
```

(a)

```
for(i=0;i<(N − 1);i++)
  for(j=0;j<M;j++)
    {U[i][j]+=V[j][i]+1;
    V[i][j]=V[j][i]-k*U[i][j];}
i=N − 1;
for(j=0;j<M-15;j++)
  {U[i][j]+=V[j][i]+1;
    V[i][j]=V[j][i]-k*U[i][j];}
mode instruction placeholder
for(j=M-15;j<M;j++)
  {U[i][j]+=V[j][i]+1;
    V[i][j]=V[j][i]-k*U[i][j];}
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    { ... ;}
```

(b)

```
for(i=0;i<N;i++)
  {if (i < M) U[i]+=V[i]+1;
  W[i]=W[i]+2;}
```

(c)

Figure 8: (a) Original code. (b) Transformed code with placeholder. (c) A code that can be loop transformed for effective low power mode usage.

The case where $M < 15$ can be handled in a similar manner. Note that precise scheduling of mode instructions is important only when we are moving from a low power mode to an active mode. If we are moving from active to a low power mode, we can issue the mode instruction immediately after the active phase. The (loop-independent) statements between

nests are treated as if they are within a (imaginery) loop nest that iterates only once. It should be noted that our approach also handles the indirect array references such as `U[X[i]]`, where `X` is the index array. In such cases, we need to consider the banks accessed by both `U` and `X`.

Our compiler-directed approach, as explained so far, has two drawbacks. First, it works at a phase granularity which means that it may miss opportunities (that might be revealed through loop optimizations) for further energy savings. For instance, consider the loop nest given in Figure 8(c), assuming that the arrays `U` and `V` reside on the same bank whereas the array `W` resides on a different memory bank. Our current approach treats this loop as a single phase assuming that both of the banks in question will be used. However, splitting the loop at iteration M would yield two loops, with the second accessing only one of the banks. Such energy-oriented loop optimizations are an interesting area for further research. Further, when an array is accessed in a nest, the current implementation conservatively assumes that all the banks holding this array are touched. Compiler support for detecting situtations where this does not hold would be invaluable. In our experiments, we explore some of the potential for detecting these situations using *programmer-defined annotations*. These annotations (which can be attached to each loop nest) provide information (to the compiler) about the *sections* of the arrays accessed in the nest, thereby enabling the compiler to detect the exact set of modules used by the nest. A similar effect can also be obtained using sophisticated compiler-based region analysis techniques [20, 42, 45].

Another potential problem is that loop splitting might incur a slight increase in datapath (processor core) and instruction cache energy consumptions due to more complex code. However, previous research [43] shows that main memory energy dominates cache and datapath energies; consequently, this is not expected to have a significant impact.

4.5 Experimental Results

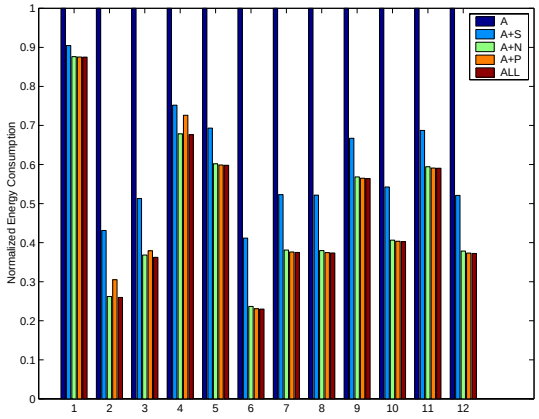

Figure 9: Energy savings due to compiler-directed mode control.

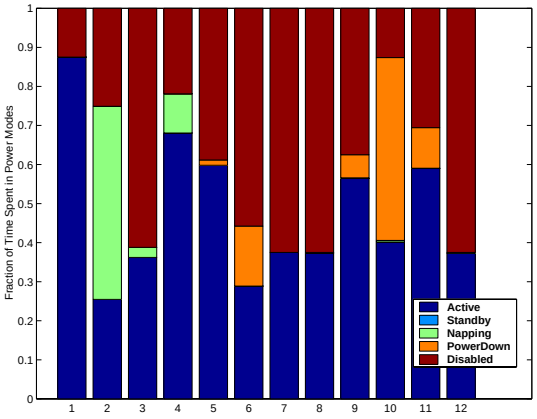

Figure 10: Relative fraction of time spent in different modes for `ALL`.

Evaluation of Compiler-Directed Mode Selection In all our experimental results (given in form of graphs), the numbers 1 through 12 represent our benchmarks (see Figure 4). Figure 9 shows the savings in DRAM energy consumption obtained from compiler-directed transitions between different operating modes. Specifically, the bar for `A+N` denotes the energy consumption if the compiler is to use only the active and napping modes, `A+P` denotes the use of only active and power-down modes, and `ALL` denotes the option for the compiler to use any of the five operating modes (i.e., active, standby, napping, power-down, and disabled). These four cases for each of the twelve benchmarks are *normalized with respect to their first bar* which denotes the power consumption if there are *no* compiler-directed operating mode transitions (i.e., the DRAMs are active at all times). As mentioned earlier, some implementations put DRAM modules [14] in standby mode soon after a reference (a compiler is not needed to perform this transition), and this is denoted by the `A+S` bar.

It can be seen that a compiler can give significant savings in DRAM energy consumption by selectively transitioning the module between the different modes. The savings range from around 12% for benchmark *adi* (1) to as much as 75% for benchmarks *dtdtz* (2) and *full_search* (6) for the ALL cases. Even if one is to compare these improvements with `A+S` which

is supported in some memories, energy savings up to 45% (an average of 23% over all applications) are achieved. *Compiler-directed transitioning to much lower energy consumption modes (napping, power-down and disabled) is thus an effective way of reducing energy consumption beyond what the current hardware does for this purpose (simply transitioning to just the standby mode).* There are two main application-related factors governing the effectiveness of compiler-directed transitions between operating modes. The first issue is whether the application can be analyzed well enough at compile time for energy optimizations. The second issue is whether the inherent application access patterns (spatial and temporal) lends itself to these compiler-directed transitions. While our benchmarks are well structured for compile-time analysis, *adi* (1) accesses arrays that span nearly all the memory banks in both of its main loop nests making the latter factor more significant in limiting the energy savings. Applications like *full_search* (6), on the other hand, have well-spaced reference patterns, increasing the scope for turning off DRAM modules that are not in use currently.

Another important observation from Figure 9 is the negligible difference in savings between A+N, A+P and ALL for each application. This can be explained by examining Figure 10 which shows the relative fraction of the total time spent by memory modules in the five different modes for the ALL execution. The energy consumption while in the active mode clearly dominates all the other modes, and the overall energy consumption is almost directly proportional to the amount of time that the modules spend in the active mode. Comparing Figures 10 and 9, one can see the proportion of time in the active mode directly materializing as the energy consumption in the ALL execution. It does *not* really matter too much what modes take up the remaining fraction of the execution time. As a reminder, the resynchronization time (which is added on to the active mode in Figure 10 and is not explicitly shown here) consumes the same amount of energy as when in the active mode. As a result, it is possible sometimes for A+P to consume more energy than A+S as in benchmarks *dtdtz* (2), *bmcm* (3), and *btrix* (4).

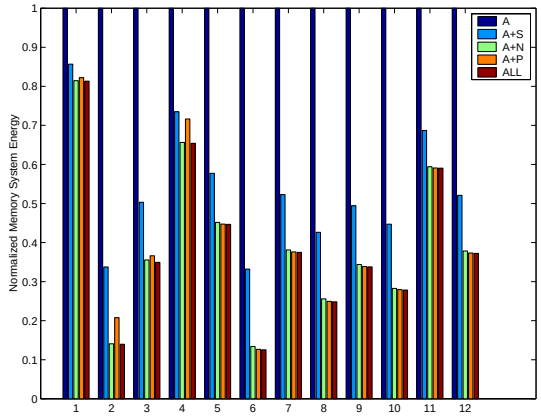

Figure 11: Energy savings due to compiler-directed mode control with clustering.

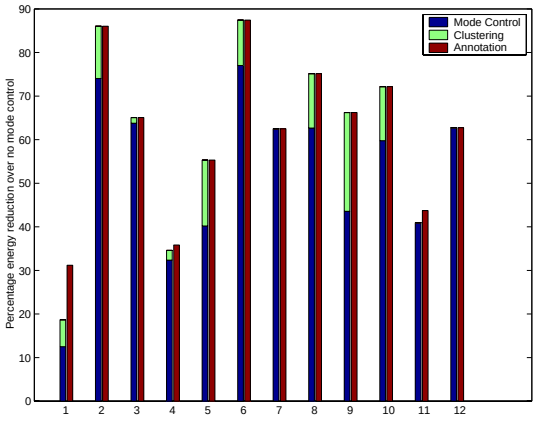

Figure 12: Impact of mode control, clustering, and programmer-directed annotations on energy savings.

Evaluation of Clustering It was mentioned earlier that clustering (allocation) of data structures by the compiler to co-locate those with similar lifetimes can boost the energy savings of mode control techniques even further. Figure 12 shows this effect by giving the percentage of energy reduction (left bar in this figure) with both mode control (ALL) and clustering compared to not effecting any transitions at all (bar A in Figure 9). This left bar is broken down into the reduction due to just mode control (without any clustering) and the extra reduction brought about by clustering. It can be seen that clustering can provide as much as 50% savings in energy over mode control (benchmarks *adi* (1) and *phods* (9)), contributing to 8% of the overall energy savings on the average over not performing any energy optimizations at all. In *adi* (1) for instance, clustering helps space out the data structures across memory modules in such a manner that at any particular time fewer modules need to be kept active than if they had not been spaced out. However, clustering is not always a useful weapon, and this can been seen with the negligible changes for *matvec* (7), *vpenta* (11), *amhmtm* (12), *bmcm* (3), and *btrix* (4). Examining *btrix* (4) closely, we observe that it references one large array spanning several modules, with the remaining referenced arrays being

relatively small. Relocating the smaller arrays to disjoint memory modules can result in additional banks being used; that is not necessarily beneficial.

It was discussed earlier how the programmer-defined annotations could be used to further the energy savings by providing specific array ranges (of the current working set) explicitly to the compiler which can then transition modules holding other parts of the array. While this can be a cumbersome task for a programmer (and compiler technology may not be sophisticated enough to detect such patterns for all programs), we have attempted such annotations for the benchmarks wherever possible. The savings with such annotations together with clustering and mode control are shown as the right bar in Figure 12. We find some energy savings with such annotations for benchmarks *adi* (1), *bmcm* (4), and *vpenta* (11). We believe that more sophisticated annotations for energy optimization (whether programmer-directed or compiler-determined) would be useful to explore further. In the rest of the experiments, we do not assume the availability of any annotations.

Figure 11 shows the energy savings due to compiler-directed mode control with clustering. From this figure, we observe that compiler support for clustering is not only important for boosting the energy savings of the compiler-directed mode control schemes (A+N, A+P and ALL described earlier), but also to amplify the savings of the A+S scheme which, as we mentioned, is done in some hardware already. Overall, the average energy reduction due to clustering and compiler-directed mode control over all twelve benchmarks amounts to around 61% (37%) as compared to employing only the A mode (A+S mode).

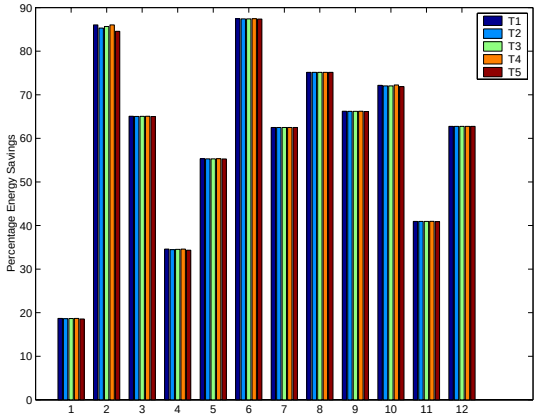

Figure 13: Energy savings with technological trends.

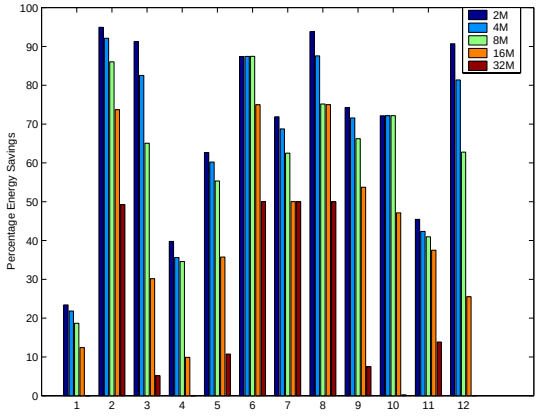

Figure 14: Energy savings with different memory organizations.

Impact of Technology It is interesting to find out how the energy savings would change with technological advancements/trends. To examine the impact of such trends, we have considered five different hardware technologies: **T1**, which is the *base configuration* with the parameters discussed in Section 2.2 on which most of the experiments in this paper are based; **T2**, which is to account for improved circuit technologies (e.g., dual threshold voltages can be used with the memory core operating at higher threshold values to reduce leakage current and the peripherals operating at lower threshold for speed) resulting in lower energy consumption (the active and standby power is set to 50% and 80% of the **T1** parameters); **T3**, which is to account for increased leakage current with very low supply voltage, hence lower threshold voltage (the energy consumptions in the napping and power-down modes are increased by 50% and 100% compared to the **T1** parameters); **T4**, which is to account for reduced synchronization times between internal and external clocks when transitioning back to active from power-down (synchronization time from power-down mode is cut by half compared to the **T1** parameters); and **T5**, which incorporates the **T2**, **T3**, and **T4** parameters.

The energy savings due to compiler-directed mode control with clustering for these five technologies are plotted in Figure 13 relative to the energy consumption in that particular technology without any mode control (only the active mode). The savings are relatively independent of the anticipated changes. As mentioned earlier, most of the energy in these enhanced executions is consumed in the active mode, and any technological improvements to cut down the energy in that mode proportionally increase energy savings with or without mode control (thus not affecting the percentage effectiveness). The

resynchronization cost from power-down mode is not really an issue as well since the compiler is able to detect the last access for memory references and is able to transition modules to disabled mode instead (and such modules will not be turned back on). This can be gleaned by going back to Figure 10 which shows that there is negligible use of power-down mode (and a much more significant use of the disabled mode). *These results reiterate the view that compiler-directed mode control with clustering is equally important across the considered technologies, and its impact is not likely to decrease as we move into the future.*

Impact of Memory Configurations All the experiments discussed until now have used a 8×1 configuration of 8 megabyte memory modules (a total of 64 MB). With increasing levels of integration (leading to a decrease in the number of modules), it is important to find out the effectiveness of the energy saving techniques. Keeping the overall memory size *constant*, one could either opt to use a larger number of smaller banks or a smaller number of larger banks. With the former option, the overall energy consumption if no energy saving techniques are employed (all banks are active) will be lower (keeping two modules active takes more energy than keeping one module of twice their size active). Further, as the number of banks gets smaller, there is less scope for turning off one or more of them. As a result, the effectiveness of the energy savings techniques are likely to diminish as modules get larger when the total size of memory is fixed. These effects are confirmed by the experimental results shown in Figure 14 which plots the energy savings for memory configurations of (a) 32×1 two megabyte modules (2M), (b) 16×1 four megabyte modules (4M), (c) 8×1 eight megabyte modules (8M), (d) 4×1 sixteen megabyte modules (16M), and (e) 2×1 thirty two megabyte modules (32M), compared to not performing mode control in the corresponding configuration. During these experiments, when we double the memory size, we have increased the energy consumption by 30%, following the trends gleaned from [35].

While the goals of smaller capacitive loads and reduced form factors are driving forces for employing memory modules with larger capacities, our results show an opposing argument for such an approach from the viewpoint of energy savings. A good design should consider the energy issues as well in deciding the memory configuration. Alternatively, it would help to provide circuitry within memory modules to selectively turn off portions of the module, or to provide more sophisticated approaches such as the ability to selectively stop refreshing at a much finer granularity (words/blocks).

5 Self-Monitored Energy Management

So far, we have only considered low power optimizations without any negative impact on performance using compilation techniques. However, this can be overly conservative since not all access information may be available/analyzable at compile time. Further, the source code for performing high level optimizations may not be even available.

We next explore a runtime approach that is referred to as the self-monitored technique since the memory system automatically transitions the idle modules to an energy conserving state. The problem then is to detect/predict idleness, and then to transition modes appropriately. However, there is the fear of misprediction with this approach that can lead to resynchronization overheads.

Use of idleness to transition an entity to a low energy mode is an issue that has been researched in the context of disks [28, 15], network interfaces [38], and system events in general [6, 7]. Some of these studies [28] have used past history (to predict future behavior) for effecting a transition. To our knowledge, no previous study has looked at this issue in the context of transitioning DRAM memory modules, where solution strategies cannot afford to incur high software and/or hardware costs to make intelligent predictions. In this paper, we explore three different hardware mechanisms for predicting inter-access times and transitioning to a low energy mode accordingly. We refer to these three mechanisms as (a) *adaptive threshold predictor (ATP)*, (b) *constant threshold predictor (CTP)*, and (c) *history-based predictor (HBP)*.

5.1 Adaptive Threshold Predictor (ATP)

The rationale behind this predictor is that if a memory module has not been accessed in a while, then it is not likely to be needed in the near future (that is, inter-access times are predicted to be long). A threshold is used to determine the idleness of a module after which it is transitioned to a lower energy mode. In ATP, the threshold is adaptive (i.e., it tries to adjust for any

mispredictions it may have made). This mechanism starts with an initial threshold, and transitions to the lower energy mode if the module is not accessed within this period. If the next access is to come soon after that (the resynchronization energy consumption is more dominant than the savings due to the lower energy mode), making the mode transition more energy consuming than if we had not transitioned at all, the threshold is doubled for the next interval. On the other hand, if we find that the next access comes fairly late, and we were overly conservative in the threshold value, then the threshold is reset to the initial value (we could try more sophisticated techniques such as halving the threshold as well).

This mechanism is employed for each degradation to a lower energy mode. Initial threshold values of 2, 100, and 1,000,000 cycles are used for transitioning from active to standby, from standby to napping, and from napping to power-down modes, respectively, for results shown in Figure 15. It should be noted that the adaptive mechanism is used only for the first transition (active to standby) in this set of results, and a constant threshold is used for the other two transitions. We observed a similar behavior while adapting the other thresholds as well. Adaptivity for the first threshold results in an average of 12% savings in energy across all benchmarks compared to fixing the threshold at 2 cycles (denoted as CTP). The ATP mechanism dynamically adapts to avoid resynchronization costs for mode changes that do not provide energy savings (as can be seen from the resynchronization times for ATP in Figure 16.)

Despite these savings, the ATP mechanism requires the calculation of energy values with the current information to decide whether to double the threshold, keep it the same, or reset it. This hardware (multipliers—for computing energies—, comparators, and a set of registers) can get complicated - consuming power as well - and, thus we do not explore this mechanism any further. Instead, one could use the ATP mechanism to decide on a good threshold value, which can then subsequently be fixed. We explore this as the next option.

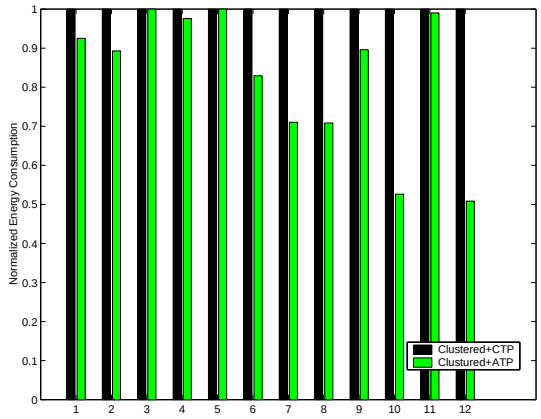

Figure 15: Energy savings with ATP.

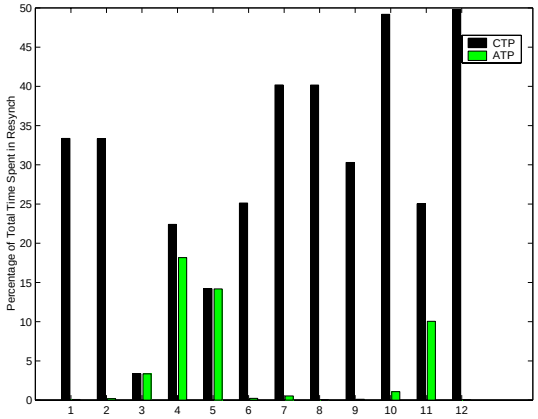

Figure 16: Resynchronization times with ATP and CTP.

5.2 Constant Threshold Predictor (CTP)

This is similar to the previous mechanism, except that the threshold is never changed (doubled). We consider this alternative mainly because of the high hardware costs of implementing the adaptive threshold mechanism. After 10 cycles of idleness, the corresponding module is put in standby mode. Subsequently, if the module is not referenced for another 100 cycles, it is transitioned into the napping mode. Finally, if the module is not referenced for a further 1,000,000 cycles, it is put into power-down mode. Whenever the module is referenced, it is brought back into the active mode incurring the corresponding resynchronization costs (based on what mode it was in). It should be noted that even if a single bank experiences a resynchronization cost, the other banks will also incur the corresponding delay.

Implementing the CTP mechanism requires a set of counters (one for each bank) that are decremented at each cycle, and set to a threshold value whenever they expire or the module is accessed. A zero detector for a counter initiates the memory controller to transmit the instructions for mode transition to the memory modules. The energy cost of this approach is significantly lower than that for the ATP mechanism.

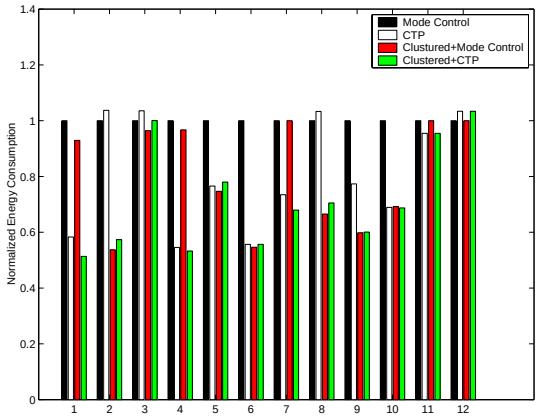

Figure 17: Energy savings with CTP.

Benchmark Number	Resynchronization Time	
	Clustered	Unclustered
1	0.07%	0.02%
2	0.13%	0.10%
3	0.02%	0.02%
4	18.16%	18.16%
5	5.42%	8.6%
6	0.22%	0.22%
7	0.53%	0.53%
8	0.01%	0.01%
9	0.11%	6.32%
10	1.07%	1.21%
11	0.05%	0.05%
12	0.01%	0.01%

Figure 18: Resynchronization time as a percentage of the execution time for CTP.

Figure 17 compares the energy consumption of the executions with (a) only compiler-directed mode control, (b) CTP, (c) compiler-directed mode control with clustering, and (d) CTP with clustering. All the bars have been normalized with respect to (a).

We find that the CTP approach gives better energy savings than the compiler-directed schemes whether the data is clustered or not for five of the twelve benchmarks (*adi* (1), *btrix* (4), *matvec* (7), *tomcatv* (10), and *vpenta* (11)). In *adi* (1), *btrix* (4), and *vpenta* (11), CTP is able to detect portions of the arrays that are not currently being worked on, which was not really possible with the compiler-directed approach we employed. This is similar to the situation we observed with respect to annotations where we had to explicitly inform the compiler as to what was currently being worked on. Self-monitoring using constant thresholds is thus able to automatically provide the annotated effect without being explicitly told by the programmer. As a result, it gives better energy savings than the compiler-directed approach which does not use annotations. In addition, CTP is also able to exploit the idleness between successive accesses to the same bank within a single loop nest that would be difficult to be analyzed statically. For example, *btrix* (4) uses a large array that spans across multiple banks. The accesses to the different portions of the array (hence the banks) are temporally staggered. Thus, banks that contain the parts of the array not currently referenced are transitioned to a lower power mode. Note that this, however, can be detrimental to performance (see Figure 18) as it takes longer to exit a lower power mode. Since, annotations only capture spatial distribution and do not account for temporal locality, more significant energy reductions are achieved using CTP as opposed to using just annotations for the *adi* (1), *btrix* (4), and *vpenta* (11) benchmarks.

In contrast, benchmarks *dtdtz* (2), *bmcm* (3), *mxm* (8), and *amhmtm* (12) perform marginally worse using CTP in both clustered and non-clustered executions. We find that in these applications the accesses are spatially and temporally local and the more conservative transitions based on thresholds employed in the compiler-directed scheme result in a slightly better energy savings. In two of the benchmarks (*phods* (9) and *tomcatv* (10)), self-monitoring gives much better results than the compiler-directed mode control when there is no clustering, and comparable performance otherwise. This can again be explained by going back to Figure 12 where we find that annotated executions of these two benchmarks give much better savings than just compiler-directed mode control (and comparable if clustering is used). Similarly, benchmarks *eflux* (5), *full_search* (6), *phods* (9) and *tomcatv* (10) do not provide much scope for more aggressive mode-control than the compiler when clustering has been employed. On an average, CTP with clustering provides 66% energy savings on the average as compared to the unmanaged power mode operation of the memory (i.e., active mode).

5.3 History-based Predictor (HBP)

There are two main problems with both previous predictors, ATP and CTP. First, we gradually decay from one mode to another (i.e., to get to power-down, we go through standby and napping), though one could have directly transitioned to the

final mode if we had a good estimate. Second, we pay the cost of resynchronizing on a memory access if the module has been transitioned. In the history-based predictor (HBP), we estimate the inter-access time, directly transition to the best energy mode, and activate (resynchronize) the module so that it becomes ready by the time of the next estimated access. This is more or less what the compiler does, except that the inter-access time is predicted here. The advantage that this mechanism has over the compiler is that once the module is back in active mode, the decay mechanism similar to CTP is used to transition to lower energy modes (in case we underestimated the inter-access time). While one could use sophisticated history information to estimate inter-access time, we use a very simple mechanism - the estimate for the next inter-access time is set to the previous inter-access time - keeping hardware implementation energy costs in mind.

HBP requires a mode assignment table that contains the maximum and minimum values of the estimated inter-access time (IAT) for which a particular mode is optimum. This table can easily be pre-constructed based on the energy values and resynchronization times for the different modes, and needs to hold only as many entries as energy modes. Once the power mode is determined, the corresponding resynchronization time is subtracted from the IAT estimate, to give the amount of time to spend in that mode. It must be noted that efficient implementations of this circuit is possible to limit the significant energy consumptions to only when mode transitions happen.

The energy savings with this approach is illustrated in Figure 19. In addition, the bars for mode control by compiler and CTP are repeated for comparison. This figure also contains a bar called 'Optimum', which denotes the optimum energy consumption that any scheme can ever hope to achieve (this can be theoretically calculated if all inter-access times are available), with respect to which the other schemes are normalized (the bar denoted as 'Integrated' is explained in the next section). We observe that while mode control by compiler consumes 48.5% more energy, on the average, than optimum, and CTP consumes, on the average, 27.4% more energy than optimum, HBP consumes only 8.8% more energy, on the average, than optimum. For nearly all but one benchmark (*btrix* (4)), HBP is very close to the optimum. In *btrix* (4), the inter-access time prediction is not very successful due to highly irregular bank accesses. We also observe that the performance penalty in the HBP scheme is generally less than that of the CTP scheme.

6 Integrated Approach to Energy Management

Until now, we have proposed and examined two approaches to effecting mode transitions - compiler-directed and self-monitored. Each of these approaches has its own advantages and disadvantages. With the compiler-directed approach, the advantage is that for the references that are analyzable, the transitions are effected exactly when required, and there are no resynchronization costs to be paid. However, there are a couple of drawbacks. First, this scheme does not lend itself very well to situations where the data structures and/or computations are not analyzable at compile time. Second, the locality of the current computation in a program may not be very apparent to dynamically transition memory modules containing portions of an array that are not currently being referenced. While annotations can be used (as was discussed), they are cumbersome for the programmer and may not always be possible. The self-monitored approach, on the other hand, enjoys the luxury of working with runtime information and can base its transitions on the dynamic locality (both spatial and temporal) of the data structures and computation. However, there is the risk of wrongly predicting the future and transitioning to a lower energy mode when one should not.

We next examine if an integrated strategy combining these two approaches can alleviate their individual shortcomings. In particular, we choose the CTP approach (which provides the mode degradation facility when not in use) and integrate it with the compiler assist for clustering, and mode control (including explicit resynchronization). The integrated strategy that we propose can be explained as follows. We can see that the compiler can do a better job than the runtime in selecting the appropriate energy mode whenever the banks are not accessed (it does not transition from one to the other as in the ATP and CTP schemes). As a result, we let the compiler-algorithm for the power setting remain the same for all the idle slots of the bank access profile. So, before an idle slot phase is executed, the compiler sets the power mode for that module, together with a bit for the memory controller to prevent any further mode switches until the compiler re-instructs it. The power back is done as before (i.e., in iterations of the previous phase) along with resetting the bit, thus allowing the CTP scheme to use its usual threshold mechanism for this module. Even if the compiler thinks that the modules may be needed, there will be

selective transition to low energy modes if this is not really the case.

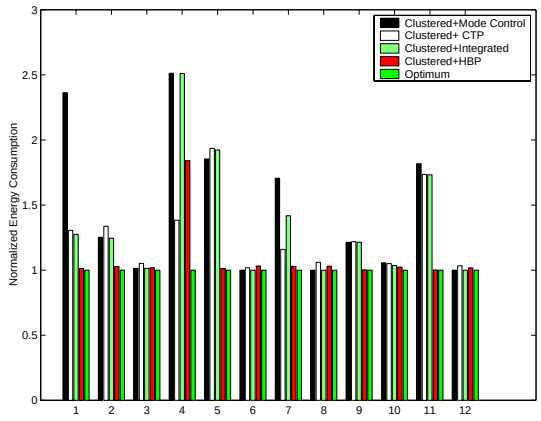

Figure 19: Energy savings comparison with optimum.

Benchmark Number	Resynchronization Time	
	Clustered	Unclustered
1	0.07%	0.025%
2	0.06%	0.0659 %
3	0%	0 %
4	0.2660%	0.0333%
5	5.14%	5.92%
6	0%	0%
7	0.03%	0.01%
8	0%	0 %
9	0%	6.19%
10	0.4495%	0.52%
11	0.0003%	0.00%
12	0%	0%

Figure 20: Resynchronization time as a percentage of the execution time for the integrated approach.

Examining Figure 19, which shows the energy consumption of the integrated mechanism relative to the others and the optimum consumption, we find that it is better than a pure compiler-directed or CTP approach for all benchmarks except *btrix* (4), *eflux* (5), and *matvec* (7). Since the banks set to a low power mode by the compiler for a particular nest have to be respected by the runtime system in the integrated approach, it may sometimes fail to exploit opportunities for further energy reduction. This is because a compiler-controlled bank in a low power mode is *always* brought back to an active state at the beginning of the nest when that bank is accessed. When there is a large loop nest and the bank is accessed only after a certain amount of time, CTP can continue to operate these banks in a lower power mode till the first access to the bank. In the other cases, the integrated approach comes very close to the best of the other two approaches. For example in *dtdtz* (2) and *vpenta* (11), it performs as well as CTP which was shown to outperform the compiler-directed approach. *Overall, the combined approach consumes only 64% of the unmanaged energy consumption of the memory system. However, we find that the HBP mechanism is able to consistently give much better energy savings than even the integrated approach, and comes very close to the optimum energy consumption.*

7 Concluding Remarks and Future Work

This paper has addressed a crucial issue in energy saving for mobile and resource-constrained computing environments (and perhaps even servers/desktops) by specifically focussing on the memory system, which has been shown [43] to consume as much as 90% of the overall system energy (excluding I/O). The paper has presented novel techniques for exploiting the low power operating modes that current and future memory technology have to offer, by detecting idle periods and transitioning the memory modules to an appropriate low energy mode without having to pay very high penalties. A set of compilation techniques to co-locate (cluster) data with similar lifetimes, and to detect idleness for mode control have been proposed. In addition, a hardware-assisted runtime approach, called self-monitoring, that can use different heuristics to predict inter-access times for mode control has also been proposed. Several prediction heuristics that can be employed for the self-monitoring mechanism have been identified. These include (a) a heuristic that uses a fixed threshold for detecting idleness and transitioning (CTP), (b) an adaptive version of this heuristic that automatically attempts to adjust to the dynamics of the program (ATP), and (c) a heuristic that uses past history to directly transition to what it estimates to be the best energy saving mode and automatically attempts to resynchronize before the next reference (HBP). It has also been suggested how the compiler-directed and self-monitoring mechanisms can be integrated. All these different mechanisms have been extensively evaluated using a spectrum of a dozen array-dominated benchmarks, to demonstrate their effectiveness.

It has been shown that compiler assistance for clustering and mode control can give as much as 85% energy savings in memory (over not performing any mode control at all), with an average saving of 61% across all twelve benchmarks. The

self-monitoring CTP mechanism gives an average energy savings of 66% across the twelve benchmarks. Between the self-monitoring mechanisms, we find HBP does significantly better than the others, and is only 8.8% higher energy consuming than the optimum mode control that one can achieve on the average. The integrated mechanism that uses compiler-directives whenever possible, and leaves it to CTP otherwise, gives an average saving of 64%. However, HBP is still better than the integrated approach in nearly all cases, because it is able to perform more global optimizations.

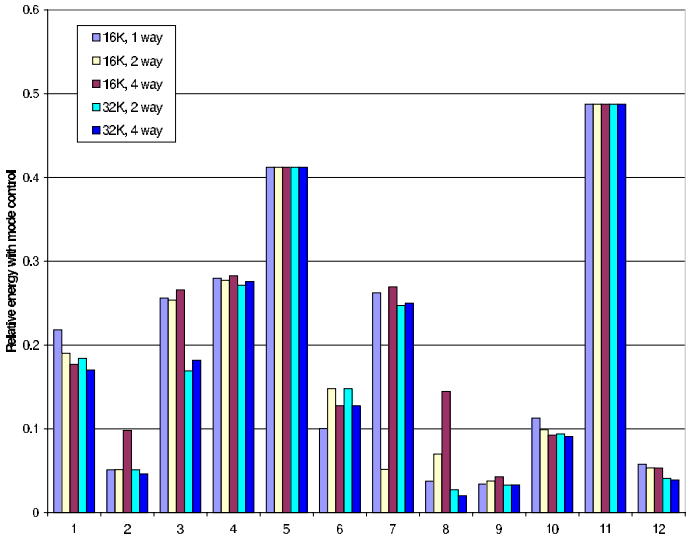

Figure 21: Relative energy consumption (with respect to no mode control) using HBP with different cache configurations.

The effectiveness of the energy savings is felt across different memory technologies and configurations, and will continue to be important as we innovate on improved circuit designs. While the evaluations in this paper have used a memory system without a cache, we have also experimented with a cache for these optimization techniques. In general, the cache tends to accentuate the influence of the mode control techniques. Due to lack of space, only the relative energy consumption with different cache configurations is given in Figure 21 for the different benchmarks with the HBP mechanism. It must be noted that HBP mechanism performed the best even in the presence of a cache.

To our knowledge, there has not been any prior extensive study looking at this wide a spectrum of hardware and software issues for energy savings of memory modules using operating mode control. Apart from the novel expedition into this important topic, the following are significant contributions of this paper that we would like to highlight:

- A compilation and evaluation framework, including energy models for different memory organizations, for studying energy-delay trade-offs.
- A set of compilation techniques for clustering of data and mode control of the memory modules where this data resides, for array dominated applications.
- A set of heuristics for making predictions at runtime to perform energy mode transitions of memory modules.

Our techniques are directly applicable to desktop or laptop environments which already have multiple memory bank organizations. While one could argue against the use/availability of multiple memory banks due to space requirements (in some mobile applications) and with deeper levels of integration, this paper suggests that one may still want to have a multiple bank organization (with mode control) from the energy viewpoint. Further, even with denser memory modules, the techniques presented in this paper can be extended for selective mode control of banks within the same chip (most current chips already have multiple banks for performance reasons).

We believe that this paper has opened a new area of exciting research for the future. The success of the proposed optimizations on array dominated benchmarks motivates the evaluation of their impact on integer and pointer dominated applications. It is also important to design new predictors accounting for the interaction with the cache. Finally, we would like to look at the impact of the virtual memory system and multi-programmed environments on the effectiveness of the proposed techniques.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In Proc. *the 32nd International Symposium on Microarchitecture*, pp. 248–259, November 1999.

[2] D. H. Albonesi. An architectural and circuit-level approach to improving the energy efficiency of microprocessor memory structures. In Proc. *the 10th International Conference on VLSI*, pp. 192–205, December 1999.

[3] Advanced configuration and power interface specification. Intel, Microsoft, and Toshiba, Revision 1.0b, Feb 2, 1999.

[4] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng The SUIF compiler for scalable parallel machines. In Proc.*the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[5] N. Bellas, I. Hajj, and C. Polychronopoulos. A new scheme for I-cache reduction in high performance processors. In Proc. *Power Driven Micro-architecture Workshop*, in conjunction with *ISCA'98*, Barcelona, Spain, June 1998.

[6] L. Benini, A. Bogliogo, S. Cavallucci, and B. Ricco. Monitoring system activity for OS-directed dynamic power management. In Proc. *ACM ISLPED'98*, Monterey, CA, 1998.

[7] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In Proc. *ACM ISLPED'98*, Monterey, CA, 1998.

[8] R. Bhargava, L. K. John, B. L. Evans, and R.Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. In Proc. *the IEEE Symposium on Microarchitecture*, Dallas, Texas, pp.37–46, Dec.1998.

[9] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards – Algorithms and Architectures*, Kluwer Academic Publishers (Boston), 1996.

[10] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In Proc. *the Fifth Intl. Symposium on High-Performance Computer Architecture*, Orlando, Jan. 1999.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In Proc *the 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, June 2000.

[12] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology – exploration of memory organization for embedded multimedia system design*. Kluwer Academic Publishers, June 1998.

[13] L-G. Chen, W-T. Chen, Y-S. Jehng, and C-T. Church. An efficient parallel motion estimation algorithm for digital image processing. *IEEE Trans. Circuits and Systems for Video Tech*, 1(4):378–385, December 1991.

[14] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., May 1999.

[15] F. Douglas, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In Proc. *Winter Usenix*, 1994.

[16] C. Ellis. The case for higher level power management. In Proc. *IEEE HotOS*, March 1999.

[17] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In Proc. *the Annual International Symposium on Microarchitecture*, December 1997.

[18] R. Gonzales and M. Horowitz. Energy dissipation in general purpose processors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1283, Sept 1996.

[19] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the desing of the Alpha 21264 microprocessor. In Proc. *the Design Automation Conference*, San Francisco, CA, 1998.

[20] P. Havlak and K. Kennedy. An implementation of interprocedural bounded region analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[21] W-M. W. Hwu. Embedded microprocessor comparison. http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[22] Intel 440BX AGPset: 82443BX Host Bridge/Controller Data Sheet, April 1998.

[23] Intel 820 Chip Set. http://developer.intel.com/design/chipsets/820/

[24] K. Itoh, K. Sasaki, and Y. Nakagome. Trends in low-power ram circuit technologies. *Proceedings of IEEE*, pages 524 –543, Vol. 83. No. 4, April 1995.

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

[26] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In Proc. *the Design Automation Conference (DAC)*, Los Angeles, California USA, June 5–9, 2000.

[27] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In Proc. *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000, *To appear*.

[28] K. Li, R. Kumpf, P. Horton, and T. Anderson. A quantitative analysis of disk drive power management in portable computers. In Proc. *Winter Usenix*, 1994.

[29] K. S. McKinley, S. Carr, and C.-W. Tseng Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[30] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In Proc. *the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[31] Pentium III Processor Mobile Module MMC-2, Datasheet 243356–001, Intel Corporation.

[32] Rambus Inc. http://www.rambus.com/.

[33] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[34] K. Roy and M. Johnson. Software power optimization. In *Low Power Design in Deep Submicron Electronics*, Kluwer Academic Press, October 1996.

[35] Samsung Semiconductor DRAM Products. http://www.usa.samsungsemi.com/products/family/browse/dram.htm.

[36] F. Jesus Sanchez and A. Gonzalez. Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS'98), Kohala Coast, January 1998.

[37] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In *Proc. DAC'99*, New Orleans, Louisiana, 1999.

[38] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, Special Issue on Mobile Computing, 2000.

[39] C. Su and A. Despain. Cache design trade-offs for power and performance optimization: a case study. In Proc. *International Symposium on Low Power Electronics and Design*, pp. 63–68, 1995.

[40] V. Tiwari, S. Malik, A. Wolfe, and T.C. Lee. Instruction Level Power Analysis and Optimization of Software, Journal of VLSI Signal Processing Systems, Vol. 13, No. 2, August 1996.

[41] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance processors. In Proc. *the Power Driven Micro-architecture Workshop in conjunction with the ISCA'98.* Barcelona, Spain, June 1998.

[42] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In Proc. *ACM SIGPLAN'86 Symposium on Compiler Construction*, Palo Alto, CA, pp. 175–185, June 1986.

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

[44] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. MICRO-29*, pages 274–286, Paris, France, December 1996.
[45] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.
[46] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In Proc. *the Design Automation Conference (DAC)*, Los Angeles, California USA, June 5–9, 2000.
[47] V. Zyuban and P. Kogge. Split register file architectures for inherently lower power microprocessors. In Proc. *Power-Driven Microarchitecture Workshop*, in conjunction with *ISCA'98*, pages 32–37, 1998.
[48] V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures, submitted to *IEEE Transactions on Computers*.