Energy Management of Virtual Memory on Diskless Devices

Jerry Hom Ulrich Kremer*

Department of Computer Science
Rutgers University

Technical Report: DCS–TR–456, September, 2001

Abstract

In a pervasive computing environment, applications are able to run across different platforms with significantly different resources. Such platforms range from high-performance desktops to handheld PDAs. This paper discusses a compiler approach to reduce the energy consumption of a diskless device where the swap space is provided by a remotely mounted file system accessible via a wireless connection. Predicting swapping events at compile time allows effective energy management of a PDAs wireless communication component such as a 802.11 or Bluetooth card.

The compiler activates and deactivates the communication card based on compile-time knowledge of the past and future memory footprint of an application. In contrast to OS techniques, the compiler can better predict future program behavior, and can change this behavior through program transformations that enable additional optimizations.

A prototype compilation system $\mathsf{EEL}_{\mathsf{RM}}$ has been implemented as part of the SUIF2 compiler infrastructure. Preliminary experiments based on the SimpleScalar simulation toolset and three numerical programs indicate the potential benefits of the new technique.

1 Introduction

Many handheld devices and machines already have wireless communication capabilities, allowing them to be part of a large and pervasive computing environment that supports sharing of resources across the network. Traditional desk-top applications will become increasingly important for handhelds which have developed from electronic address books and appointment schedulers to portable workstations. For instance, the newest Compaq iPAQ H3600 handheld has 64MB of RAM, 16MB of flash memory, and a 206MHz low-power StrongARM processor [8]. Such devices will run spread-sheets, voice and image recognizers, and even computation intensive simulation programs, just to mention a few. However, many mobile machines may not have secondary storage such as a disk. Giving mobile machines the ability to support virtual memory through a wireless connection can significantly increase their functionality since the same programs can be executed on a desktop machine and the handheld. This is particularly important for programs where the memory needs vary significantly based on the provided input data. However, the option of swapping pages over the wireless connection comes with the price of additional energy requirements due to the wireless networking card and communication costs. In this paper we discuss a compilation strategy that will reduce the energy overhead of swapping over a wireless network through network card hibernation.

Resource hibernation is an effective strategy to save power and energy of system components and resources that are not needed during some parts of a program execution. While not in use, these components and resources consume energy which may be saved by transferring them into a hibernation or sleep state during their idle periods. System resources may implement different levels of hibernation, where each level has a specific tradeoff between power saved vs. the time it takes to deactivate or reactivate the resource. Typically, the "deeper" the hibernation or sleep mode, the longer it will take to make a transition to and from this hibernation state, but the less power will be used by the resource during the hibernation period. Effective power and energy management of a wireless connection is crucial for handheld devices that rely on battery power since the communication component typically consumes a substantial share of the overall energy and power budget. On Compaq's iPAQ H3600 pocketPC, communication via an Orinoco WaveLAN 802.11b wireless card consumes more than 40% to the overall energy budget of an image processing application [14].

*{jhom,uli}@cs.rutgers.edu; This work was partially supported by NSF CAREER award No. CCR-9985050. To appear in Proceedings of the Workshop on COLP'01.

The ACPI (Advanced Configuration and Power Interface [9]) standard specifies hibernation states for different system resources such as disks, wired and wireless Ethernet controllers, processors, and displays. ACPI conforming systems are possible target systems for our compilation strategy. Most work in resource management for power and energy savings purposes has concentrated on operating system and hardware techniques [11, 6, 15, 16, 19]. In this paper, we investigate the potential benefit of compiler directed resource management for a system resource such as a wireless communication card. We will also compare our approach with an OS approach where deactivation is based on a threshold strategy, and activation is done on demand. Our benefit study is based on a set of three numerical, array based applications (*shal*, *adi*, and *tomcatv*). All three applications represent *regular* problems, for which many program characteristics can be derived at compile time. In the future, we will consider irregular problems and pointer based programs. We believe that computation intensive simulation codes will be part of the program mix for portable workstations such as Compaq's iPAQ pocketPC.

In this paper, we assume that only a single application is executing on the handheld machine. In a multiprogramming environment, the information collected by our compiler can be used by the underlying operating system to effectively schedule page requests across different active processes.

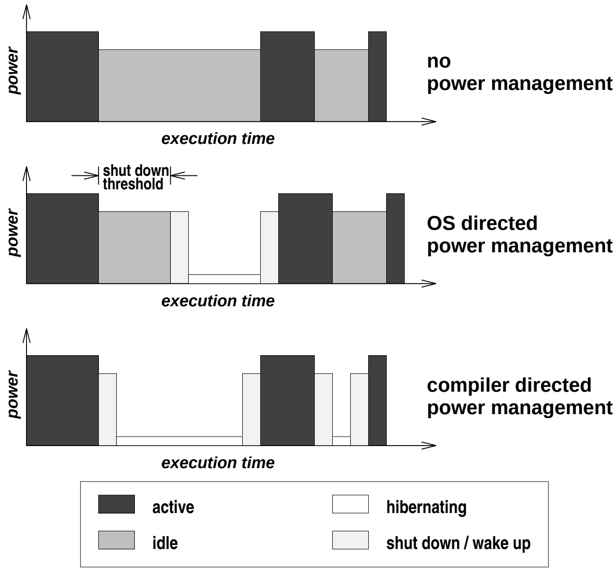

Figure 1: Comparison of compiler vs. OS directed power mangement.

2 Related Work

The idea of remote virtual memory, particularly distributed and/or shared, has been an ongoing subject for over 15 years. Comer and Griffioen examine the usefulness of a dedicated memory server in [7]. They make the distinction of separating the paging operation from the file backing store operation. Then they can focus on designing efficient memory and file servers. Another approach views the sum total memory of a cluster as a single cache space [10]. Dahlin et al. suggest utilizing the memory of idle nodes. These approaches improve performance by optimizing the use of extended virtual memory.

Recognizing the utility of remote resources, Schilit and Duchamp make the case for thin clients [18]. They conclude the feasibility and desirability for thin clients without a disk and smaller amounts of memory. While not necessarily studying energy consumption impacts, their work establishes a reference point in motivating low power designs of diskless devices. From a compiler point of view, we attempt to optimize energy demands by managing resources such as virtual memory paging.

3 Problem Formulation

For simplicity, we assume that a communication card only supports three power mode states: active, idle, and sleep (hibernate). In the active mode, the card is transmitting data. In idle mode, the card is not sending messages, but listens to the wireless networking traffic. Finally, in the sleep or hibernation mode, the card has been shut down to save power. There is an overhead for transitions between hibernation modes. We assume that the performance penalties for shutting down and waking up the card are the same.

Figure 1 shows the power profile of a sample application without any power management, with operating system guided, and with compiler-directed power management. The simple OS based technique transfers the card into sleep mode after a predefined (static) inactivity threshold. The wake-up operation is performed on demand, and as a result incurs a performance penalty.

This simple example illustrates the advantages of a compiler-directed approach vs. a threshold based OS approach. In the former approach, system resources can make the transition into power saving states earlier, can be reactivated just-in-time to avoid performance penalties, and enable additional optimization opportunities for idle periods which are shorter than the threshold used by the OS based technique. It is important to note that there are more sophisticated OS based dynamic power management techniques than the simple technique discussed here [15, 19]. However, the point we want to make is that in many cases the compiler can predict future program behavior and resource requirements more accurately than OS based techniques, allowing additional opportunities for power and energy management optimizations.

The handheld device is connected to a network file system (NFS) via the wireless connection. Each time a page fault occurs, the required page has to be requested over the wireless link, and the program blocks until the page is received. Each page fault event leads to a new working set, with the empty set as the initial working set of an application.

Our compilation strategy tries to identify program parts of the program execution where the working set is either

1. the same for the next x machine cycles, or
2. is about to change in y machine cycles.

This information is used to suspend the wireless card if x is larger than a predetermined benefit threshold, or resume the card in y cycles, where y is the time needed to reactivate the card. Both entities will be determined by the compiler using static performance prediction.

OS guided hibernation may use threshold techniques to shut down system components such as a wireless card. Threshold techniques assume that if a resource has not been used within the past *threshold* time units, it will not be used in the future.

4 $\mathsf{EEL}_{\mathsf{RM}}$ Prototype Compiler

The $\mathsf{EEL}_{\mathsf{RM}}$ [1] prototype compiler is based on the SUIF2 compilation infrastructure [1]. The compilation strategy consists of two main phases, with each phase having multiple steps. During the first phase, program regions are identified for which the wireless connection needs to be activated or deactivated. The data objects accessed in each region are summarized, and a forward data flow problem approximates the data objects that will be in memory before entering each region. If the set of data objects that will be referenced in a region is a subset of the data objects currently in memory, the execution of the region does not require the wireless connection to be active.

[1] EEL stands for Energy Efficiency and Low-power, and RM stands for Resource Management. Information about the EEL laboratory can be found at http://www.cs.rutgers.edu/∼uli/eel.

In the second phase, system calls are inserted that either activate or deactivate the wireless PC card. Deactivation is done as soon as possible, and activation is performed on-demand. Activation and deactivation operations are assumed to be atomic, i.e., once the PC card is in the process of being shut-down, a pending wake-up request has to wait until the shut-down has been completed and vice versa. The second phase requires performance prediction for efficiently placing activation requests. An activation request before a program region should only be executed if the card is in a hibernation state. If the card is active, no action needs to be taken. This can be easily handled through the activation routine itself, or through compiler generated guards for each activation or deactivation request.

Performance prediction is needed to activate the PC card just in time. For instance, if the overhead of activation is 10^6 cycles, the activation request should be issued 10^6 cycles before the card needs to be active. In addition, performance prediction is required to assess the benefit of deactivating the PC card. Deactivating the card is not beneficial if the next activation request follows too closely (i.e., before the card is shut-down, a request to reactivate it is already pending).

4.1 Phase 1 - Analysis

This analysis phase consists of several subtasks.

1. Program regions are identified that will serve as the basis for our analysis. The compiler will insert hibernate or activate instruction only before such regions. The initial prototype system recognizes innermost loop nests, called phases [12], and calls to runtime system functions (e.g. `printf`) as program regions. $REGIONS$ denotes the resulting set of regions. The region control flow graph (RCFG) has $REGIONS$ as its set of nodes, with edges representing the possible control flow between these regions. The RCFG is similar to the phase control flow graph (PCFG) introduced by Kennedy and Kremer [12].

2. Initially, data objects are scalar variables and arrays with their declared sizes. For instance, subcomponents of arrays, such as single rows and columns in the two-dimensional case, are not considered. For each region $r \in REGIONS$, two sets of data objects d are determined:

   (a) $d \in MUST\_REF(r)$, if d is referenced during every execution of region r;

   (b) $d \in MAY\_REF(r)$, if d may be referenced during an execution of region r;

The $MUST\_REF$ sets are used to describe data objects that will be in memory after the execution of the corresponding region, and MAY_REF sets are the basis to predict future data object references that may require swapping over the wireless connection.

3. The data flow problem $IN\_MEM(r)$ is solved. For each entry point of a region r the set of data objects that are in memory is determined. Since cache policies such as LRU keep track of the sequence of data references within a finite window of past references, a notion of time or *decay* has to be incorporated into the data flow formulation. Initially, we will solve this problem by a simulation process.

4. Each region r is labeled as *yes* or *no* depending on whether the region may require swapping over the wireless connection or not.

$$\text{if } MAY\_REF(r) \subseteq IN\_MEM(r)$$

$$\text{then no, otherwise yes}$$

4.2 Phase 2 - Code Generation

The compiler inserts calls to runtime routines $activate$ and $hibernate$. The effect of these routines are

$$activate \Leftrightarrow \begin{cases} \text{system call "card\_on"} & \text{if card is inactive} \\ \text{no action} & \text{if card is active} \end{cases}$$

$$hibernate \Leftrightarrow \begin{cases} \text{system call "card\_off"} & \text{if card is active} \\ \text{no action} & \text{if card is inactive} \end{cases}$$

The initial approach will place calls to $activate$ and $hibernate$ at region entry points. A limited set of reshaping transformations to enable additional optimizations will be considered. Performance prediction will be used to move $activate$ statements up the region control flow graph to program points that allow the overhead of the activation to be overlapped with program execution.

Performance prediction will also be used to eliminate $hibernate$ statements that are considered unprofitable due to subsequent $activate$ operations. If the distance in terms of execution cycles between a $hibernate$ and $activate$ operation is too close, the benefit of shutting-down the card is lost. A backward-flow, $\forall$-information data flow problem can be used to determine the length of the minimal $activate$-free path for any region exit point. Hoisting of $activate$ operations, and elimination of $hibernate$ operations may be done in a combined analysis pass.

```
    float A(n), B(n), C(n)
R1  do i = 1, n
        A(i) = ...
    enddo
R2  do i = 1, n
        B(i) = ...
    enddo
R3  do i = 1, n
        C(i) = ...
    enddo

R4  do i = 1, n
        B(i) = ... C(i) ...
    enddo
R5  do i = 1, n
        A(i) = ... B(i) ...
    enddo

R6  print A
```

Figure 2: Sample code

Our initial benefit analysis assumes that the compiler performs a reshape optimization called page fault clustering. Assuming that swapping operations are atomic, i.e., cannot be overlapped, this transformation will not impact the overall performance of the program. Page fault clustering is applied if the memory footprint of a region ($MAY\_REF(region)$) fits into memory. Prefetch instructions are generated before such regions, allowing all potential page faults to occur before the execution of the region, leaving the region free of page faults. This transformation allows potential hibernation of the communication card during the entire region execution.

4.3 Performance Model

For each region, the performance model has to report the number of cycles needed to execute it. In our initial system, symbolic entities such as program size and loop bounds are assumed to be known at compile time. We will use a micro-benchmarking approach to determine basic computation and memory access costs as well as the suspension and activation time of the wireless communication card [4, 17, 13].

At a later point, we will consider parameterized (symbolic) performance expressions. Our analysis and code generation strategy has to be modified in order to allow the evaluation of these expressions at runtime, and based on the results, will executed the guarded $activate$ and $hibernate$ operations.

4.4 Example

region	memory size		
	4	8	12
R1	miss	miss	miss
R2	miss	miss	miss
R3	miss	miss	miss
R4	miss	no miss	no miss
R5	miss	miss	no miss
R6	miss	no miss	no miss

Table 1: Page faults for different memory sizes in terms of pages, assuming that each array requires 4 pages of memory space.

In the example program shown in Figure 2, we assume a memory size of 4, 8, and 12 pages, a write-allocate paging strategy, and a LRU page replacement policy. The array size n is set such that each array occupies 4 pages. To simplify the example, scalar variables are ignored, and arrays are assumed to be aligned at page boundaries. Table 1 lists the data space page faults expected to occur for different memory sizes.

Whether a card should be shut down for a region that does not incur a page fault will depend on the predicted execution times for the region. For example, if it takes longer to shut down the card than executing regions R4 or R6, then it is unprofitable to shut down the card for these two regions for the 8 page memory case. However, for the 12 page memory, shutting down the card will be profitable.

4.5 Implementation Issues

For our initial implementation, we started with a simple memory access model to see how closely we approach actual behavior. In simplifying the memory access, we assume an entire array will be loaded (used) whenever there is a reference to it. By examining the array's declared size and data type, we calculate the number of required memory pages. However, there are instances where only a single row/column is accessed, or the array is accessed in a triangular pattern. In such cases, we will need more accurate tools to analyze memory patterns. We plan to use a modified form of Data Access Descriptors (DADs [5, 3]).

Using DADs can aid our analysis in two ways. First, DADs describe an iteration order in walking through the dimensions of an array. As pages are swapped out after a given loop, we may reasonably estimate which pages of an array remain in memory. For instance, one loop may iterate forward over an array, while another loop may iterate backward over the same array. It can be safe to assume the last x pages of the array are still in memory. Secondly, DADs also help by more accurately indicating the accessed regions of an array. If only a single row/column is needed, then the array's memory access summary is given by the necessary page(s), and the overall loop memory block summary will be more concise.

The current prototype implementation approximates LRU. Our LRU simulation strategy does not consider virtual addresses, but instead uses data and code access summary information. For each region, a single data structure describes all data and system calls (`printf`) referenced in the region. In addition, the total number of pages needed to store all data and code in memory is recorded.

A key component for approximating LRU is the notion of age. Along with summarizing array accesses at a region level, we associate a relative age for each region. Hence, all array accesses within a region have the same age and will be replaced at the same time. This is easily represented in a queue, where each element is the region summary information. In addition, we can remove elements from anywhere in the queue. For example, if a referenced array is found somewhere in memory, the containing region is removed and placed at the end. If a region is larger than the total memory, the net effect is to clear the contents in memory.

The current implementation computes $MAY\_REF(r)$ for each region r. Instead of computing separate $MUST\_REF(r)$ sets, we set $MUST\_REF(r) := MAY\_REF(r)$, which is a simplification. The solution to $IN\_MEM(r)$ is approximated by applying the LRU simulation process to nodes in the RCFG, starting with the entry node, and choosing the next node according to the rPOSTORDER numbering [2]. The initial value of $IN\_MEM(r)$ is $\emptyset$. If a loop is encountered, its entire body is visited twice. The resulting values in $IN\_MEM(r)$ represent the final solution for region r. This process is applied recursively for nested loops. Our heuristic is motivated by the observation that the stable state typically occurs after a loop has iterated at least twice. The heuristic may lead to visiting sequences exponential in the loop nesting depths. However, the maximal loop nesting depth in a program is typically a small constant. Our current implementation always picks the most frequently executed branch of a conditional statement as the only branch that is ever executed.

Although we have used and made several simplifying assumptions, our analysis is able to predict most page faults correctly. Table 2 shows the total number of correctly predicted hits and misses (True Hit/Miss) as well as incorrect predictions (False Hit/Miss). The parameters used in these benchmarks are listed in Table 3. The page size is assumed to be 4KB. Only in the case of *tomcatv*, the False Miss count was significant. The misprediction oc-

	shal	*adi*	*tomcatv*
True Hit	17	62	304
True Miss	9	1	304
False Hit	1	0	2
False Miss	2	0	100

Table 2: Dynamic page hit/miss prediction accuracy.

Parameters	*shal*	*adi*	*tomcatv*
N	32	16	32
M	32	16	16

Table 3: Benchmark parameters.

curred for a rather small region, resulting in no significant impact on the overall energy savings and performance. Detailed energy and performance results are given in the next section.

5 Experiments

We modified the SimpleScalar simulator to keep track of page faults that occur during the execution of a program. In addition, the simulator logs the cycle times where program regions such as loops are entered and exited. The simulator allows the assessment of the amount of computation performed for a given working set, and the resulting potential benefit of suspension and resumption of the wireless card.

For three different programs, we evaluated the working sets for different memory and program sizes. Given a particular overhead of the suspend and resume operation (25,000 CPU cycles), we determined the performance impact and energy savings of our optimization.

If working sets change frequently, the wireless card should never be suspended. If the working sets are changed very infrequently, both OS and compiler based approaches will lead to similar results. Compiler techniques are superior to OS techniques in cases where a working set does not change for a length of time that is comparable to the OS based suspension threshold and on-demand resumption times.

We assume a performance predictor tells us which regions take longer than the time required for a suspend operation and then use on-demand resume. We compared the potential energy savings of our compiler techniques vs. OS static inactivity thresholds strategies of varying lengths. Through ACPI, the OS allows the user to tune threshold levels for various devices. Therefore, we use thresholds relative to the suspend operation time (suspend overhead).

From simulation traces, we have a notion of time (cycle counts) for each benchmark. We also have a correlation of system power consumption given that earlier measurements show the WaveLAN card to consume 40% of total system power (iPAQ + WaveLAN). However, hibernation mode still draws a small amount of power. Therefore, while in hibernation we consider total power demands to drop by 1/3. The power level actually drops a little more than 1/3, so this is a slightly conservative over-estimate (which also makes calculations easier!). Translating this into energy comparisons is just a summation or integral under the curve of the power levels across execution (cycle) time.

5.1 Benchmark Characteristics

In *shal*, there are few regions which access the same arrays consecutively across loops. Conversely with *adi*, each loop uses all arrays; hence there is one large region to suspend the card (after the arrays have been loaded). We see more interesting behavior in *tomcatv* where there are some opportunities to suspend within a large loop (containing several nested loops), yet the entire loop does not fit into memory. Thus, each iteration should use some power management policy to save overall energy.

These three benchmarks use two dimensional arrays of size $N \times N$. We chose sizes of N along with the number of memory pages M that exhibited interesting behavior. Each memory page is assumed to have 4KB. If M is too large, then after initial array accesses there will be no more page faults. If M is too small, arrays may not fit at all, requiring page faults everywhere. Adjusting N mainly affects the simulation execution time, therefore we try to keep it small. The parameters used in these benchmarks are as shown in Table 3.

We want to use OS inactivity thresholds relative to the suspend operation, however we have measured both suspend/resume times to be about 130ms under Linux 2.4 for the iPAQ, which amounts to about 25 million cycles. For interesting benchmark results, this requires simulation runs on the order of days. In order to reduce simulation times, our analysis scales this factor by 1000 before calculating potential energy benefits (i.e., assumes a more ideal situation in terms of suspension/resumption overhead).

OS threshold	EEL_{RM} Energy Results			
	shal	*adi*	*tomcatv*	*tomcatv (PFC)*
1×	101.0	99.3	126.5	95.3
10×	100.1	92.6	116.3	87.6
20×	99.7	86.2	104.2	78.5
24×	**99.4**	—	—	—
30×	99.7	80.6	98.6	74.3
35×	99.7	78.1	**96.7**	**72.9**
54×	99.7	**69.1**	96.7	72.8
∞	99.7	71.3	96.7	72.8

Table 4: Energy consumption of benchmark programs with EEL_{RM} energy management. Energy figures relative to OS approach.

OS threshold	OS and EEL_{RM} Performance Results		
	shal	*adi*	*tomcatv*
1×	101.3	101.7	105.4
10×	100.3	100.2	103.0
20×	100.2	100.2	102.9
24×	**100.3**	100.2	—
30×	100.0	100.2	101.0
35×	100.0	100.2	**101.5**
54×	100.0	**103.2**	100.0
∞	100.0	100.0	100.0
EEL_{RM}	100.2	101.7	101.0/103.9 (PFC)

Table 5: Execution times relative to the ∞ threshold.

5.2 Simulation Results

Table 4 shows the effectiveness of our compilation strategy over an operating system approach which is based on inactivity thresholds for card suspension. The reported figures assume a 25,000 CPU cycles suspension overhead. Results for different OS threshold values are listed, where each such value is a multiple of the suspension overhead of the wireless communication card. The ∞ threshold represents the case where the communication card is always on (i.e., never suspended). Boldface numbers indicate the points at which longer thresholds have equivalent energy/performance characteristics as the ∞ threshold.

Comparing a range of thresholds reveals subtle results. Picking an optimal value is impossible because program behavior is unique. Furthermore, small changes to the threshold, in hopes of tuning energy/performance, have negligible impact. A slight change may allow the OS to hibernate during an additional region, but possibly at the cost of incurring a performance penalty at another region. Conversely, adjusting the threshold to avoid a performance penalty may prohibit the OS from hibernating at another region. An example of this behavior was observed in *shal*. From 1x – 10x, energy usage vacillated while performance improved marginally.

Overall, the results show that for *shal*, the OS technique and our EEL_{RM} compiler perform roughly equivalent with little, if any, energy savings. Our compiler does a better job against larger thresholds in the *adi* case due to the fact that it is able to suspend the card earlier. Since there is one large suspended region, the compiler's advantage grows linearly with respect to varying the threshold limit. This results in energy savings of almost 30% over the OS based technique.

For *tomcatv*, our compiler does not perform well compared to short OS threshold values. This occurs because of computationally large loops which contain page faults. Our compiler identifies this and keeps the card enabled. Therefore, we miss large opportunities for hibernation. However, we examined page fault clustering as an enabling optimization. By swapping in all necessary data before a regions, the compiler can direct the card to hibernate within the region. In the presence of page fault clustering (*tomcatv (PFC)*), our approach always does better than the OS approach, with energy savings of up to 27%. In all cases, the compiler based approach reduced the energy consumption of all benchmarks as compared to the case without any power management.

Note that there is an implicit asymptotic limit of the energy savings attainable by power managing the wireless card (i.e., shutting down the card immediately after program start and for the entire duration). For the case of a WaveLAN on iPAQ, the energy savings limit is about 33%. Indeed, results from *adi* show our technique reaches 28.7% savings. On the opposite extreme, we cannot do much for *shal*, but neither can the OS. In programs exhibiting behavior similar to that of Figure 1, *tomcatv* reveals the potential for more intelligent power management through those idle periods than the OS.

Although the reported results were obtained for small problem and main memory sizes, we expect the results to scale well if both entities grow proportionally ($N^2 \propto M$). However, if the problem sizes grow faster than the main memory sizes, enabling transformations such as page fault clustering and index set splitting will become increasingly important for effective compiler-based techniques.

Using a power management approach may lead to performance degradation due to the on-demand resumption penalty of the wireless card. A summary of the overall performance penalties is given in Table 5. The largest penalty we observed for EEL_{RM} was 3.9% relative to the program performance without any power management. Overall, the performance penalties can be considered insignificant.

6 Future Work

Clearly, more advanced analysis techniques and experiments are needed to further validate the effectiveness of our approach. Our current implementation does not use a performance model to eliminate *hibernate* statements or perform just-in-time card activation. We are in the process of integrating page fault clustering as an enabling transformation into our compiler. In addition, we will consider index set splitting as an additional enabling transformation in cases where the working set of a region is too large to fit into memory.

We are planning to extend our method to consider explicit file I/O, irregular applications, and programs with pointer-based data structures. We will investigate how much improvement over our current approach can be achieved by using refined DAD-based implementations for MAY_REF, $MUST\_REF$, and solving the IN_MEM data flow problem. We are also planning to apply our techniques to non-scientific applications such as browsers, voice recognition, and image understanding codes.

7 Conclusion

Compiler-directed energy management of a wireless communication card can be an effective strategy as compared to an OS based energy management approach. Simulation results showed energy savings of up to 30% over the OS. For OS inactivity threshold of 10x – 20x card suspension overhead, energy savings improvements of up to 21.5% were observed, assuming that page fault clustering was applied to enable energy optimizations. Not only do these results show potential energy benefits, but we also wish to emphasize, even under adverse conditions, our compiler does not perform significantly worse than the OS. That is, our analysis tries to ensure actual energy savings before directing the wireless card to hibernate.

Although our intent is to show the benefits and feasibility of compiler techniques, our results provide an interesting guide for ACPI. In general, smaller thresholds yielded modest energy gains with little performance delay. This can be understood by noticing that

$$\textit{execution time} \gg \textit{resume overhead}$$

Our preliminary estimates in eliminating this performance delay by assuming just-in-time activation provide up to an additional 5% energy savings.

References

[1] National Compiler Infrastructure (NCI) project. Overview available online at http://www-suif.stanford.edu/suif/nci/index.html., Co-funded by NSF/DARPA, 1998.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, MA, second edition, 1986.

[3] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, Williamsburg, VA, April 1991.

[5] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[6] T. Burd and R. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing*, 13(2-3):203–222, 1996.

[7] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proc. Summer 1990 USENIX Conf.*, pages 127–126, Anaheim, CA (USA), 1990.

[8] Compaq Corp. *iPAQ H3600 handheld PC*. http://www.handhelds.org/Compaq.

[9] Intel Corp., Microsoft Corp., and Toshiba Corp. ACPI implementers' guide. Draft, February 1998.

[10] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 267–280, Monterey CA (USA), 1994.

[11] S. Devadas and S. Malik. A survey of optimization techniques targeting low power VLSI circuits. In *Proceedings of the 32th Design Automation Conference*, 1995.

[12] K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, 1998.

[13] U. Kremer. Fortran RED – a retargetable environment for automatic data layout. In *Eleventh Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, NC, August 1998.

[14] U. Kremer, J. Hicks, and J. Rehg. A compilation framework for power and energy management on mobile computers. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, August 2001.

[15] J. Lorch and A. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3), June 1998.

[16] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *IEEE Trans. on Computer Aided Design*, 17(11), November 1998.

[17] R. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, February 1992. UCB/CSD-92-684.

[18] B. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. Technical Report CUCS-004-91, 1991.

[19] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Dynamic power management for portable systems. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, August 2000.