Compiler And Runtime Support For Predictive Control Of Power And Cooling

Henry G. Dietz and William R. Dieter
Electrical and Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
{hankd, dieter}@engr.uky.edu

Abstract

The low cost of clusters built using commodity components has made it possible for many more users to purchase their own supercomputer. However, even modest-sized clusters make significant demands on the power and cooling infrastructure. Minimizing impact of problems after they are detected is not as effective as avoiding problems altogether. This paper is about achieving the best system performance by predicting and avoiding power and cooling problems.

Although measuring power and thermal properties of a code is not trivial, the primary issue is making predictions sufficiently in advance so that they can be used to drive predictive, rather than just reactive, control at runtime. This paper presents new compiler analysis supporting interprocedural power prediction and a variety of other compiler and runtime technologies making feed-forward control feasible. The techniques apply to most computer systems, but some properties specific to clusters and parallel supercomputing are used where appropriate.

1. Introduction

At all levels, power and cooling issues are becoming critical factors in computer design and operation. The problem with power and cooling is that, unlike most other performance criteria, power and cooling are integrative system-level effects with relatively poor measurement facilities, long time constants, and significant hysteresis.

Commercially available processor and motherboard designs generally do not provide sensors that can measure energy or power (energy consumed per unit time). Some laptop computers employ intelligent batteries that can provide rough readings, and external meters also can be used, but these measurements are averaged over relatively large intervals and convolve processor power use with that of other components. The runtime system may be able to note which components are contributing to the current power consumption, but distinguishing the individual contributions is difficult and accuracy is further degraded by the process.

More targeted thermal sensors exist on many motherboards, but thermal sensing can take as long as 5 ms to 10 ms and there is significant hysteresis. For example, temperature at a sensor may continue to rise long after power consumption at the heat source has been dramatically reduced. One of the authors has an old laptop that senses temperature to control a fan so that a maximum operational temperature is never exceeded, but the fan stops immediately when power is turned off. The annoying result is that the laptop frequently overheats just *after* being turned off, and a thermal interlock will not let it power on again until the temperature has dropped to be within range... which can take quite a while without a fan running. It is precisely this type of hysteresis that makes *predictive* thermal management necessary: by the time a problem is directly sensed, it is usually too late to correct it.

Any annotation of code with energy consumption estimates must support predicting behavior far enough into the future to be useful in the timescale in which the system will be controlled by the runtime system. That timescale ranges from hundreds of microseconds to seconds: very far into the future given a processor that executes thousands of instructions every microsecond. Standard engineering practice provides neither compiler nor runtime technologies able to hoist predictions about power and thermal properties of a region of code so far in advance of the code's execution.

In Section 2, we present a methodology by which a compiler (or an assembler, linker, or loader) can use static analysis of instruction-level energy use to construct power and cooling predictions looking many thousands of instructions ahead. Techniques for efficiently transmitting the predictions to the runtime control and a framework using this predictive ability in support of multiple version coding also are

1-4244-0054-6/06/$20.00 ©2006 IEEE

presented. Section 3 describes a method for creating or refining predictions at runtime, as well as guidelines for runtime use of predictions. The contributions are summarized in Section 4.

2. Compiler Techniques

An increasing number of compiler researchers are focusing on development of compiler technology that treats power consumption as a first-class performance parameter for compilers to optimize [6, 13]. Ironically, the compiler technology we advocate in this paper does not promise any major advances in compiling code to use less power. Our compiler contribution centers on converting instruction-level energy estimates into a mechanism efficiently supporting *predictive* control.

At the instruction level, compile-time estimation of power consumption is possible using any of a wide variety of approaches [17, 4, 15]. Many of the techniques discussed in the literature use detailed architectural models, but such models are difficult to create and maintain for the processors and other subsystems commonly used in cluster nodes. The difficulty is further compounded by the fact that different revisions of a part often have different power attributes and documentation of architectural details is not always freely available. Thus, we prefer to use an instruction-level accounting method that is based on empirical measures. Such a technique also can account for costs associated with specific system calls, which defy architectural modeling.

In order to provide higher-resolution power information at runtime, Bellosa et al [2, 3] propose a methodology in which a calibration technique is used to associate power costs with specific performance counter events (e.g., cache misses) that closely correspond to causes of power changes but can be sampled faster. Compiler technology for predicting many of these performance counter events is fairly mature; for example, Chi and Dietz were performing sufficiently detailed cache miss analysis in 1989 [5]. It is even simpler to use calibration, perhaps by fractional factorial experiments, to associate energy consumption with static program attributes that might not be tracked by hardware performance registers – such as energy consumed by each type of instruction. For cluster computing, we also can take advantage of the inherent homogeneity of the system; it is feasible to augment a single prototype node with specialized, potentially expensive, current and/or thermal probes to obtain calibration data that will be highly accurate for a cluster of identical, but uninstrumented, nodes.

For the purposes of this paper, it is sufficient to appreciate that there are many viable ways to determine upper bound, lower bound, and expected (average) energy consumption for each instruction. Combining that data with pipelined scheduling logic such as that already in most compilers trivially yields good energy, execution time, power, and heat estimates for a basic block of code at a time. Prediction could thus be as simple as inserting an instruction posting the predicted value(s) at the start of each basic block. The problem is that basic blocks are generally far too small to be a useful unit of prediction for power and thermal control at runtime.

```
main(...) {
   A g(...); B g(...); C
}
g(...) {
   D if (E) { F g(...); G } H
   return;
}
```

Figure 1. Sample Recursive Program

```
main:  A goto g;
x:     B goto g;
y:     C goto exit;
g:     D if (!E) goto t;
        F goto g;
z:     G
t:     H on ... goto x, y, z;
```

Figure 2. `goto`-Converted Recursive Program

2.1. Prediction Lookahead Analysis

A basic block contains no control flow: every instruction will be executed if any instruction is executed. Predict longer-term future behavior is fundamentally similar to the analysis needed to perform very deep speculative execution: the difference is that instead of hoisting code to be speculatively executed earlier, we are summarizing power and thermal properties over the possible future execution paths and inserting code to post those summaries to the runtime system. The approach we have discovered and present here can handle prediction across arbitrary control flow without any direction from the programmer.

In 1999, Dietz[7] developed new compiler analysis and transformation techniques to support speculative predication across arbitrary control flow, e.g., in support of the predicated execution facilities of the Itanium architecture. In fact, that technique was very similar to a method Dietz developed in 1993[8] and has continued to use for direct conversion of a MIMD program into an equivalent SIMD program. The first step in both older techniques and the new approach proposed here is to convert the program into a state transition graph in which each basic block is a node and arcs between nodes represent guarded (predicated) control flow. Function call and return, even for recursive functions, is thus represented without any distinction; a call creates an arc and a return creates multiple arcs, one to each of the possible return positions.

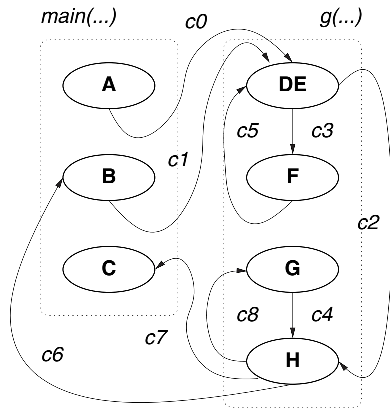

Figure 3. Recursive Program's State Machine

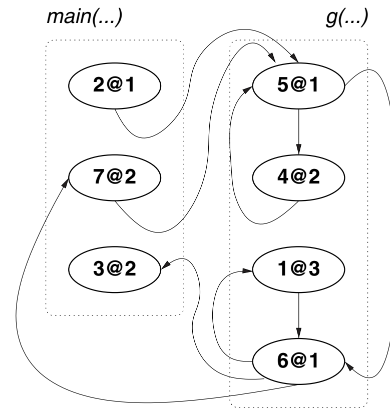

Figure 4. Timing@Power Labeling

For example, Figure 1 shows the outline of a simple C program containing a recursive function call. A function call is really just a `goto` accompanied by some data manipulation; function `return` is essentially a computed `goto` based on data saved when the function was called. This `goto`-conversion yields code roughly structured like that listed in Figure 2.

The resulting state transition graph is shown in Figure 3. This graph has precisely the same structure described in Dietz [7]. Each arc is labeled with a condition upon which that arc is traversed, however, for simple timing prediction analysis we need not consider these conditions. The graph serves primarily as a way to simplify reasoning about future behavior. Only the power, timing, and thermal attributes of the basic block within each state are critical.

As suggested earlier, there are many ways to compute approximate energy consumption, execution time, power, heat generation, etc., for individual instructions, and hence for basic blocks. Each of these quantities can be estimated as an expected (average) value or as minimum and maximum bounds. Further, if the calibration process is detailed enough to identify where heat is generated (within the physical processor chip layout), it may be useful to track heat as a vector with components corresponding to heat output in each identifiable portion of the physical system.

For our example, consider labeling each state with the simplest possible prediction information: expected power consumption and expected execution time. Suppose that analysis determines that basic block **A** in Figure 3 takes 2 units of time to execute at an average power consumption of 1 unit of energy per unit time. We will denote this as simply **2@1**. Labeling all nodes in the graph in this way yields the graph in Figure 4.

Although the labels in Figure 4 are predictions if the information is posted at the start of execution of each state, they are unlikely to predict behavior far enough in advance. The primary reason for building this graph is so that predictions can be extended over longer time periods that are more meaningful in terms of the time constants encountered in runtime power and thermal control, thus providing useful predictions with minimal overhead. Thus, consider collecting a prediction for each node that looks T units of time ahead, where T is determined by the runtime predictive control parameters, with typical values for T in the range of thousands to millions of processor clock cycles.

Conceptually, the state machine model of a program is similar to Nondeterministic Finite Automata (NFA) used to describe lexical recognizers. However, lexical analyzers are normally built from Deterministic Finite Automata (DFA), not NFA, so a conversion process is applied. In the context of speculative predication [7], MIMD to SIMD conversion [8], or of the power and cooling prediction discussed here, we refer to any such transformation as Meta-State Conversion (MSC): construction of an equivalent meta-state graph that deterministically incorporates information from multiple original states in each meta-state. The MSC rules used for each purpose are essentially closure operations, but with very different semantics that result in very different graphs. The closure process used to label a state machine with T-unit lookahead predictions of maximum average power is simpler than for the other purposes, leaving the graph structurally unchanged. The per-node prediction lookahead MSC is described in Algorithm 1; computing predictions for all N nodes requires only O(TN) effort.

Closures to determine predictions for other attributes work the same way. For example, determining the minimum average power with T-unit lookahead would substitute "minimum" for "maximum" in step 3. A variety of special-case optimizations can be made to improve the accuracy and speed of the basic algorithm. For example, loops with compile-time known iteration counts can be analyzed

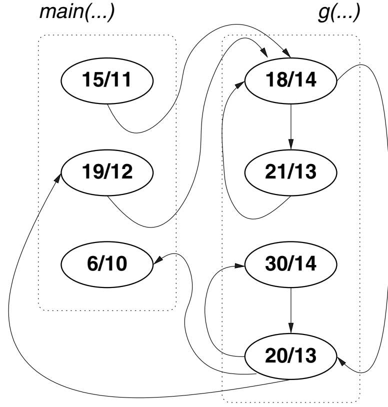

Figure 5. T=10 Power/Prediction Interval

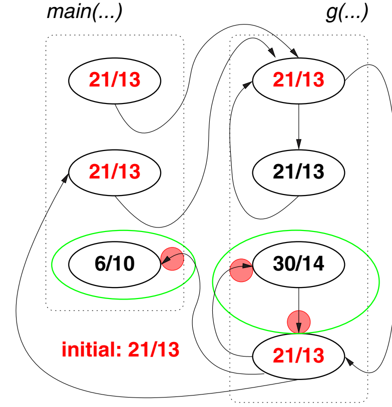

Figure 6. E=0.4 Prediction Posting Operations

Algorithm 1 T-Unit Lookahead For Maximum Power Prediction

---

1. Determine all paths rooted at the given node which (1) have expected execution time no shorter than T and no longer than one node past an expected execution time length <T or (2) prematurely reach a terminal node (such as **C** in Figure 3).

2. For each path, compute the sum of the products of the node execution time and power (i.e., the total energy expended in the node) and divide that product by the sum the node execution times. This yields average power for the path.

3. Label the root node with the maximum average power value computed for any of the paths.

---

much as though they consisted of a single properly-labeled node. It also is possible to significantly prune the path enumeration in step 1 by retaining intermediate results and recognizing when a path reaches a node that has already been visited. In any case, the expense of the analysis occurs entirely at compile time.

Although the example in Figure 4 is too small to yield an interesting labeling for a realistically large T, the average power estimates for a period of T=10 units of time in the future are shown in Figure 5. We have kept the predicted power expressed as fractions to clarify how they are derived by computing power divided by path length. For example, the labeling of **A** as **15/11** is because the path **A-DE-F** yields (1*2+1*5+2*4) / (2+5+4) = **15/11** whereas the path **A-DE-H** yields (1*2+1*5+1*6) / (2+5+6) **= 13/13**. Note that the path lengths used are generally a fraction of a node execution time longer than T. For typical values of T, the error is negligible; alternatively, it is possible to examine the instruction-level predictions within the last state in each path to obtain the precise value of T desired.

2.2. Encoding To Minimize Runtime Post Cost

Although state machine node labeling described above trivially provides all the T-unit lookahead predictions that a predictive runtime controller might want, the mere act of posting a prediction at entry to each state would constitute significant execution overhead. If posting was implemented by inserting an instruction at the start of each basic block to store a constant prediction value into a memory location visible to the runtime control system, not only might the overhead reach double-digit percentages of runtime, but the posting operations themselves could cause the predictions to be wildly inaccurate – because the energy consumed by the posting operations was not modeled. The energy of posting could be accounted, but a more efficient method for transmitting predictions to the runtime controller is a better answer. There are two basic ways in which the accuracy of the predictions accessed by the runtime controller can be maintained with lower overhead:

Reduce the frequency of posting. AbouGhazaleh, Childers, et. al [1] proposed an algorithm to insert power management points into a program; our goal is to insert fewer posting events in the code. This can be accomplished by selecting an error threshold, E, such that any error in the prediction which is less than E can be ignored. Algorithm 2 attempts to minimize the number of prediction posting operations needed while maintaining a specified maximum error. Applying our algorithm with E=0.4 to the example of Figure 5 results in the graph shown in Figure 6, which results in a reduction from 7 to just 3 prediction posting points (the small shaded circles) and no increase in the final number of states (the prediction posting states are all merged into existing states). Note that, if multiple attributes are to be predicted, the algorithm would be applied separately to reduce posting operations for each attribute.

Algorithm 2 Insertion Of Prediction Posting operations

1. Let P_S be the prediction in state S. Augment every state S with a posting value, V_S, initialized as V_S=P_S. Also mark every state as "unprocessed."
2. While there exists an "unprocessed" state X:
   (a) Mark X as "processed."
   (b) For each arc A, that goes from state X to some state Y:
      i. If (V_X> V_Y) then Z=Y and M=V_X else Z=X and M=V_Y
      ii. If ((V_X!= V_Y) AND ((M-E) < P_Z)) then V_Z=M and state Z is marked as "unprocessed."
3. Statically initialize the posting location to the value V_S where S is the start state. This value serves as the initial estimate before the program has begun to execute.
4. For each state Y such that there exists at least one arc A that goes from some state X to Y such that V_X!=V_Y:
   (a) Construct a new state, Z, that contains only the code to post the prediction V_Y.
   (b) For each arc A that goes from some state X to Y such that V_X!=V_Y, replace arc A with an arc from X to Z.
   (c) Insert an arc from Z to Y.
5. Perform traditional code straightening to merge any pair of states X and Y such that the only arc leaving X is also the only arc entering Y.

Use demand sampling rather than posting. Rather than having the code actively post predictions, it is possible to encode all predictions within a static data structure that can be accessed on demand by the runtime predictive control. The most obvious mechanism would be to build a map of the program code that would allow the runtime control logic to use the current program counter (PC) from the process to index the appropriate prediction from the map. The map has many redundancies; many PC values yield the same prediction, so it is likely that the lookup table can be dramatically compressed using compressive hashing [16]. Fundamentally, the idea is to find a hash function by which PC values that hash to the same hash table entry have the same, or very similar, prediction values. The hash function could be compiled-into the process as a signal handler; whenever a new prediction is needed, the runtime control logic would signal and the handler would respond by hashing the PC valued saved when the signal handler was invoked and posting the prediction thus recovered. If multiple attributes are to be predicted, they can be tupled in the hash function or obtained using multiple hash lookups.

Algorithm 3 User-Supplied Multiple Version Syntax

```
#powercase
/* code for default algorithm */
#poweralt
/* code for first alternative */
#poweralt
/* code for second alternative */
...
#poweresac
```

2.3. Multiple Version Encoding

Multiple version encoding is a well-known compiler technique used to create multiple alternative codings for a construct such that the runtime system can dynamically select the best one to execute. Most often, multiple version encoding was used to select between parallel and serial algorithms based on the result of a runtime dependence check, but we also can use this technique to select between codings with different power profiles. These alternative codings could be provided by the user or automatically created by the compiler.

Given a method for the runtime control to provide the user program with a "contracted" level of power consumption, it would be very simple for a user to mark power-contract-based branch points in their program. For example, suppose that a particular functionality could be implemented by any of several alternative algorithms that the programmer suspects have significantly different power and thermal profiles. Without the programmer knowing the performance of each routine, the programmer could tell the compiler to encode all alternatives and ask the compiler to evaluate the properties and use the runtime contract to determine which version to execute.

A simple syntax might be as shown in Algorithm 3. The compiler would treat the `#powercase` construct as a forced position for posting predictions for all the alternatives, generating code that evaluates the runtime system's contract to conditionally jump to the alternative that best matches the contracted power and thermal profile.

The fully automatic generation of power variants would work in much the same way. After identifying specific code sections that have well-known transformations into variants with potentially different power, the compiler would treat the variants just as though they had been provided by a user with the construct above.

For example, if there are two coded versions of a meta-state (collapsed prediction region), a slow one requiring 50W-60W and a fast one requiring 75W-80W, the compiler prediction might say 50W-80W. If the runtime system came back with a 70W contract, the compiler would force the slow alternative coding to be used. In this way, the compiler is allowing the operating system to exert relatively fine-grain predictive control within each process –

not just across multiple processes. This distinction is important because, unlike data centers, parallel supercomputers have good reason to be running very few programs in a timeshared mode. Without the ability to adjust trade-offs within a process, there might not be enough processes for the operating system to throttle between.

It is worth noting that a compiler could err, or a devious programmer could lie, giving the runtime system a prediction that is far less resource use (power) than the code requires. However, by being somewhat conservative, the operating system can detect such a problem before critical limits have been exceeded and throttle by not scheduling that code. Thus, the worst case is the same behavior that other, non-predictive, power management approaches seek.

3. Runtime System Software

The compile-time analysis provides an excellent starting point for predictive control at runtime, but having good power estimates at compile time is not the same as being able to effectively use predictive control at runtime. The first problem is selection of an appropriate mechanism for passing compile-time estimates to the runtime system. Secondarily, as good as compile-time estimates may be, there is potential improvement available by combining static compile-time estimates with historical records of dynamic behavior at runtime. This is especially true if malicious users have caused false (low) predictions to be posted in the hope of obtaining more than their fair share of the computing resource. Indeed, it might be feasible to forgo the compile-time analysis entirely if the runtime history mechanisms are sufficiently effective. Finally, it is important to make good use of the predictions; we will not propose a complete system here, but merely provide a few insights and guidelines as to how predictions should be used.

3.1. Runtime History-Based Predictions

As powerful as the above compile-time mechanism is, it would be nice to be able to improve upon the estimates by using runtime history. There are many examples of hardware structures designed to predict branching behaviour based on history; what we propose is a fully software-managed *Predictive Power History Buffer (PPHB).*

As suggested above, power is not easy to measure directly. The closest approximation to direct measurement is the indirect calibration of the power cost associated with various types of events that are easily accounted, as per Bellosa et al [2, 3] – the same general approach that we favor for the compile-time analysis. Another alternative is to use relatively slow-responding thermal sensors. Temperature while executing a region of code does not necessarily have any direct correlation with power consumption or heat generation by that code region; temperature might be quite high and rising despite the code currently being executed having a relatively modest power profile. There are two reasons:

Temperature is an integrative measure. A recent history of high power consumption might make the temperature have a high average over the region despite execution of the region significantly cooling the processor. It is thus far more correct to look at the temperature change rather than the temperature. Of course, simply recording simple differences ignores the basic thermodynamic fact that higher temperatures require more energy to be maintained, so the best accuracy will be obtained by calibrating a scaling factor for temperature differencing.

Temperature changes with significant hysteresis. The temperature change caused by one region of code might not be visible until long after that region has completed executing. How long after? Again, only a calibration process can produce the best accuracy.

There are several ways in which the history buffer can be organized. Perhaps the most obvious would be to borrow the same type of hash-indexing commonly used for hardware BHBs (Branch History Buffers). In this type of organization, the instruction address associated with a sample would be hashed and the resulting data is stored in that line.

The primary difference between PPHB and BHB operation is that the PPHB is not simply recording an unambiguous fact about code uniquely identified by the PC value; rather, it is intended to be predictive of power consumption over an extended period after the PC was at the address recorded. We envision a signal handler within the program being awakened at regular intervals that are short relative to the minimum period of runtime predictive control operations; we will call the integral sampling rate multiplier R, with typical values between 2 and 10. In addition to the PPHB itself, an auxiliary FIFO data structure is needed. The basic signal handler algorithm is given in Algorithm 4.

Algorithm 4 PPHB Signal Handler

1. Recover the program counter (PC) from the interrupted process and compute the power consumption since the last sample using techniques such as those proposed by Bellosa et al [2, 3]. Insert the pair in the FIFO buffer as PC_t, P_t.
2. Compute the power prediction P by averaging P_t, P_{t-1}, P_{t-2}, ..., $P_{t-(R-1)}$.
3. Update the history in PPHB[hash($PC_{t-(R-1)}$)] using P. The update may consist of replacing an existing entry or averaging with it, modifying minimum, average, and maximum values, or updating more complex data structures.

The primary difficulty with this procedure is that the PPHB will not be filled with good predictions very quickly.

Smaller PPHBs fill faster, but also have a higher probability of interference. An interesting possibility is to use the static compiler analysis to initialize the PPHB and then refine the static predictions using dynamic measurements.

3.2. Control Algorithms

Rather than simply switching power management on and off as the cluster approaches its thermal limit, the runtime system should use control theory to smoothly apply power management to hold the system at maximum performance without exceeding thermal limits [19]. In addition to using feedback control, long-term power predictions from the executing code can add a feed-forward term based on the expected reaction of the system, thus tracking the desired operating point more closely. Although predictive thermal management has not been used for parallel systems, it has been shown to be more effective than reactive management for uniprocessor multimedia applications [20]. Effective control allows over-provisioning, i.e., building a bigger computer than can be continuously fully powered [9].

Power reduction can be implemented using any of the standard mechanisms discussed in the literature, such as dynamic voltage and frequency scaling (DVS), instruction fetch throttling, and choice of code sequences. Compiler-generated multiple version encoding (see Section 2.3) offers additional control of power vs. speed tradeoffs within a process. Switching frequency and voltage implies an overhead of up to 500 μs, whereas runtime selection between alternative codings based on a power contract can be done with overhead measured in nanoseconds.

The techniques introduced in this paper fit well with a wide range of existing control techniques [21, 18, 10, 11, 14, 12]. The compile-time and/or runtime power predictions trivially improve the effectiveness of any reactive control method by allowing control actions that had been initiated reactively with very conservative thresholds to instead be initiated using predictions with significantly more aggressive thresholds. The multiple-version coding, especially used in combination with paired minimum and maximum power predictions, also makes it possible to cheaply implement some control by simply setting contracts.

Of course, new control laws emphasizing feed-forward control should be able to do even better. We envision a power management control law that not only takes full advantage of power predictions as described here, DVS, and multiple-version coding, but also of runtime modeling of environmental issues ranging from Computational Fluid Dynamics (CFD) for predicting heat flow within the room that houses a cluster computer to issues of electricity pricing and cooling system fluctuations due to external conditions (lower efficiency on a hot day, a building's cooling system being turned-off on weekends or cool days, etc.). We are working toward developing this level of integrated modeling and control.

Traditionally, computers have had very little hardware and software implementing "autonomic" self-evaluation and control. Deliberately making a system that will normally have to run below peak speed due to power and thermal issues also goes against common supercomputing sense. However, we view these investments as enabling the system to get better performance when power and cooling are less expensive or more available than than in the worst case, thus maximizing value. For example, the electricity cost for operating the KASY0 cluster supercomputer for one year exceeds the cost of its network hardware; over the lifespan of the machine, it will exceed the cost of the processors! Thus, a cluster whose performance is limited by the cost of power and cooling during on-peak hours can run much faster during off-peak hours when electrical costs are lower and air conditioning is more efficient due to lower outdoor temperatures, yielding better overall price/performance.

4. Conclusion

Although there is a large and growing body of work aimed at managing power and cooling attributes of a computing system, the critical difference between these attributes and those that have been successfully managed in the past is that these require dealing with long time constants and hysteresis. Feed-forward control, not just reactive feedback, is needed to deal effectively with the long time constants and hysteresis associated with power and thermal control. Most work in this field has suffered from using measurements taken at that moment or over the recent past as though they were a prediction of future behavior.

In this paper, we have presented new compiler and runtime technology that can efficiently create, and make accessible to runtime control, true predictions of behavior for arbitrarily long periods in the future. The compiler and runtime technologies described here are very flexible, and can be used for large-lookahead predictions of many kinds. It remains to be seen precisely which power and thermal attributes will be most useful in implementing predictive runtime control, nor do we yet know how much better the control will be using true predictions of future behavior as input. Our future work in this area is thus focused on implementing a variety of these new techniques and obtaining experimental results to guide further development.

On a larger scale, we see power and thermal issues becoming significant components of all aspects of system design, programming, and operation. We already have modified the Cluster Design Rules (CDR) software tool to model power and cooling constraints and operating costs when designing a cluster computer. We also see a pressing need to

model complete environmental issues ranging from Computational Fluid Dynamics (CFD) models of heat flow within the room that houses a cluster computer to issues of electricity pricing and cooling system fluctuations due to external conditions (lower efficiency on a hot day, a building's cooling system being turned-off on weekends or cool days, etc.). In each case, control using prediction of future circumstances is the key to getting the best performance.

References

[1] Nevine AbouGhazaleh, Bruce Chiders, Daniel Mossé, Rami Melhem, and Matthew Craven. Energy management for real-time embedded applications with compiler support. In *ACM SIGPLAN Joint Conference LCTES'03*, June 2003.

[2] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 37–42, New York, NY, USA, 2000. ACM Press.

[3] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[4] L. N. Chakrapani, P. Korkmaz, V. J. Mooney III, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: What can a (poor) compiler do? In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 01)*, pages 176–180, November 2001.

[5] Chi-Hung Chi and Henry G. Dietz. Unified management of registers and cache using liveness and cache bypass. In *PLDI*, pages 344–355, 1989.

[6] Keith D. Cooper and Todd Waterman. Understanding energy consumption on the c62x. In *Workshop on Compilers and Operating Systems for Low Power (COLP 02, co-located with PACT 02)*, Charlottesville, Virginia, USA, September 2002.

[7] H. G. Dietz. Speculative predication across arbitrary interprocedural control flow. In *Languages and Compilers for Parallel Computing: 12th International Workshop, LCPC'99*, volume 1863, pages 432–446, London, UK, June 2000. Springer-Verlag.

[8] H. G. Dietz and G. Krishnamurthy. Meta-state conversion. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II, pages 47–56, August 1993.

[9] Mark E. Femal and Vincent W. Freeh. Safe overprovisioning: Using power limits to increase aggregate throughput. In *Workshop on Power-Aware Computer Systems*, December 2004.

[10] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 11, 2005.

[11] Chung hsing Hsu and Wu chun Feng. Power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 9, 2005.

[12] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 9, 2005.

[13] Masaaki Kondo, Shinichi Tanaka, Motonobu Fujita, and Hiroshi Nakamura. Reducing memory system energy in data intensive computations by software-controlled on-chip memory. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP 02), co-located with PACT 02*, September 2002.

[14] Ramakrishna Kotla, Soraya Ghiasi, and Freeman L. Rawson III Tom W. Keller. Scheduling processor voltage and frequency in server and cluster systems. In *IPDPS 2005*, page 8, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Tao Li and Chen Ding. Instruction balance and its relation to program energy consumption. In *Languages and Compilers for Parallel Computing: 14th International Workshop, LCPC 2001*, pages 71–85, August 2003.

[16] Muthulakshmi Muthukumarasamy and Henry Dietz. Empirical evaluation of compressive hashing. In *Workshop on Compilers for Parallel Computers*, January 2006.

[17] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Instruction-level power estimation for embedded vliw cores. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 34–38, New York, NY, USA, 2000. ACM Press.

[18] Ratnesh K. Sharma, Cullen E. Bash, Chandrakant D. Patel, Richard J. Friedrich, and Jeffrey S. Chase. Balance of power: Dynamic thermal management of internet data centers. *IEEE Internet Computing*, 9(1):49–49, January–February 2005.

[19] Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2002.

[20] Jayanth Srinivasan and Sarita V. Adve. Predictive dynamic thermal management for multimedia applications. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 109–120, New York, NY, USA, 2003. ACM Press.

[21] Andreas Weissel and Frank Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the First Workshop on Temperatur-Aware Computer Systems (TACS'04)*, Munich, Germany, June 2004.