Cache-aware Optimization of BAN Applications[*]

Lei Ju, Yun Liang, Samarjit Chakraborty, Tulika Mitra, Abhik Roychoudhury
Department of Computer Science, National University of Singapore
{julei, liangyun, samarjit, tulika, abhik}@comp.nus.edu.sg

Abstract

Body-area sensor network or BAN-based health monitoring is increasingly becoming a popular alternative to traditional wired bio-monitoring techniques. However, most biomonitoring applications need continuous processing of large volumes of data, as a result of which both power consumption and computation bandwidth turn out to be serious constraints for sensor network platforms. This has resulted in a lot of recent interest in design methods, modeling and software analysis techniques specifically targeted towards BANs and applications running on them. In this paper we show that appropriate optimization of the application running on the communication gateway of a wireless BAN and accurate modeling of the microarchitectural details of the gateway processor can lead to significantly better resource usage and power savings. In particular, we propose a method for deriving the optimal order in which the different sensors feeding the gateway processor should be sampled, to maximize cache re-use. In addition, we also investigate the effects on cache re-use of different memory layouts of the code processing the different sensor data. The joint optimization of code layout and the order in which the different sensors should be sampled – in order to maximize code cache re-use – turns out to be a difficult combinatorial optimization problem. But our experiments show that optimizing the sampling order of the sensors has a much larger influence on cache re-use, compared to the effects that different code layouts have. Based on this, we also propose a heuristic that obtains near-optimal solutions in jointly optimizing both code layout as well the sensor sampling order. Our case study using a *faint fall detection* application – from the geriatric care domain – which is fed by a number of smart sensors to detect physiological and physical gait signals of a patient show very attractive power consumption in the underlying processor. Alternatively, our method can be used to improve the sampling frequency of the sensors, leading to higher reliability and better response time of the application.

1 Introduction

Body-area sensor networks (or BANs) and related wearable computing technologies have lately become very popular, particularly in the context of biomonitoring applications. Well-known projects and prototype architectures in this area include MIThril [4], CustoMed [11], Wearable

[*]A preliminary version of this paper appeared in the Proc. International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS) 2008, and was nominated for a best paper award.

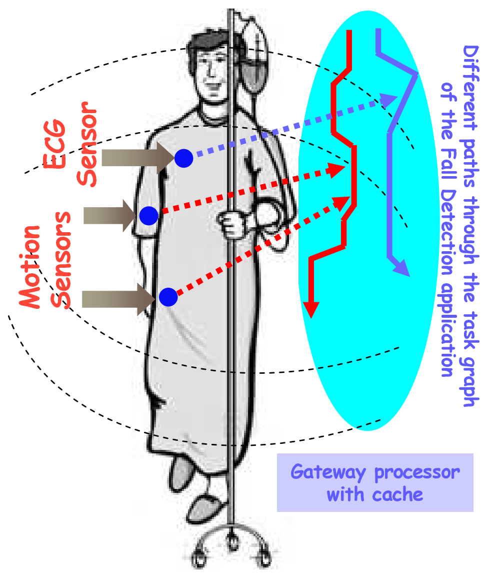

ECG Sensor
Motion Sensors
Different paths through the task graph of the Fall Detection application
Gateway processor with cache

Figure 1: Data from different sensors triggering different parts of the application on the gateway processor.

e-Textiles [5], Wearable Motherboard [18], e-Textile [12], and RFab-Vest [10]. Growth in this area has been largely fueled by the recent technological advancements in embedded processors, availability of lightweight sensor nodes, and advances in wireless networking. As a result, BAN-based health monitoring is increasingly becoming a viable alternative to traditional wired biomonitoring techniques, which require a patient to be hospitalized and hooked up to large monitoring equipments.

However, most biomonitoring applications require continuous processing of large volumes of data streams arriving through multiple sensors. As a result, both computation bandwidth and power consumption turn out to be serious constraints while designing sensor network-based computing platforms for high-end biomonitoring applications. This has led to a trememdous interest in architectures, design methods and software analysis techniques specifically targeted towards wireless BANs and wearable-computing related applications such as biomonitoring (for example, see [6, 7, 11, 13] and the references therein).

Our contributions: In this paper we propose a disciplined method to optimize applications running on the communication gateway of a wireless BAN by exploiting the microarchitecture of the processor and the code layout of the application in memory. More specifically, we show how to compute the optimal order in which the different sensors feeding data into the gateway processor should be sampled in order to maximize the cache re-use. Techniques for optimal code/data [17, 20] placement to maximize cache re-use – which are related to the problem we address in this paper – have been well-studied in the embedded systems literature along with techniques for optimally configuring a cache [8]. But the problem of optimizing applications processing data from multiple sensors has not received sufficient attention so far. An important difference between well-known cache-aware code/data placement techniques and the problem we address here is that the former case is primarily concerned with static data. On the other hand, we are concerned with data *streams*, where different data items trigger different parts of the application code.

As mentioned above, computing the optimal sampling order of the sensors – *given* a code layout in memory – is closely connected to computing the optimal code layout to maximize the instruction cache re-use. Hence, we also consider the problem of *jointly* optimizing the layout of the code processing the different sensor data, and the order in which the different sensors should be sampled. As expected, such a joint optimization leads to a combinatorial explosion even with relatively small problem sizes. To get around this, we have investigated – with one concrete real-life application – the relative influences of code layout and sensor sampling order on the instruction cache re-use. It turns out that at least for the application at hand, the sensor sampling order has a significantly larger effect on cache re-use compared to different code layouts. Based on this observation, we propose a heuristic for jointly optimizing both sensor ordering as well as the code layout. Our heuristic gives near-optimal solutions and we believe that the basis for this heuristic also holds for a wide variety BAN applications.

We have applied our method to a real-life *fall detection* application from the geriatric care domain, which is fed by eight different smart sensors connected to a patient's body. By using a combination of physiological and physical gait signals, this fall detection application is able to distinguish between regular motion (e.g., walking and stair climbing) and the onset of an imminent fall. Once the possibility of such a fall is detected, various precautionary measures can be activated or an alarm might be triggered to call for help. Hence, the application has strict real-time requirements; an increased sampling rate of the sensors increases the reliability and responsiveness; and finally — as with all other applications in this class — it should incur low battery energy consumption. Clearly, these goals contradict each other and complicate the design of both the underlying hardware and the software, especially when combined with additional requirements like light-weight, ease of use, and small form-factor. Hence, this calls for careful optimization and co-design of the hardware and the software. Our results show that the order in which the eight sensors are sampled in each round (which does not affect the functionality of the application) has a significant effect on the degree of instruction cache re-use and hence the processing time of the sensor data. Ignoring this sampling order can lead to up to 24.8% increased average power consumption per clock cycle in the gateway processor. Alternatively, the reduction in processing time can be exploited to increase the sampling rate of sensors, which leads to improved reliability and responsiveness of the application.

The above power savings were obtained with a "natural" code layout (i.e., one that was obtained by compiling the code written by the application developer). By jointly optimizing the code layout as well as the sensor sampling order, a further 3% reduction in the average power consumption was obtained. This shows that while code layout does have an impact on instruction cache re-use (and hence power consumption), its influence is significantly lower than that of different sensor orderings.

Overview of the proposed method: A high-level overview of our setup is shown in Figure 1. Data from multiple wireless sensors attached to a patient's body arrive at a battery-operated gateway processor (also strapped to the patient), which runs various biomonitoring applications implementing fall detection, electrocardiogram (ECG) analysis, etc. Often such applications are run on regular personal mobile devices such as mobile phones or PDAs. However, given that such devices usually have large form-factors and short battery life, there is an increasing trend towards building customized gateways that have been tailored for specific biomonitoring applications. We assume such an application-specific gateway having a lightweight processor (running at 60 - 80 MHz) with a small instruction cache.

Depending on the sensor from which the data arrives, different parts of the application code are triggered. However, often there is a significant overlap between the codes corresponding to two different sensors. As the processing time of any sensor data depends on the state of the instruction cache (that is determined by the previously executed code blocks), the specific order in which data from different sensors is processed has a significant impact on the number of instruction cache misses and hence the processing time. Towards this, our cache modeling relies on fixed-point construction-based analysis of program semantics [16] and is similar to the ones used for estimating the so-called *cache-related preemption delay* (see [15]). When the number of sensors in question is relatively small, it may be feasible to exhaustively enumerate all possible sensor orderings, compute the processing time corresponding to each of these orderings by accurately modeling the instruction cache states, and finally select the ordering with the smallest processing time. However, this scheme breaks down when the number of sensors start increasing. For high-end biomonitoring applications, the number of sensors can easily exceed 20 (e.g., 12 sensors for ECG monitoring [13] plus 8 sensors for our fall detection application). For these many sensors, it is easy to see that exhaustively enumerating all possible sensor orderings can be ruled out. For such problems, we rely on modeling the instruction cache re-use from the last one sensor only. In other words, given an ordering of sensor nodes $\ldots, S_i, S_j, S_k$, when estimating the maximum execution time of the code for S_k, only the cache state resulting from processing S_j's data is accounted for; we ignore the cache state resulting from processing the data from S_i and all previous sensors. Although this leads to some (safe) over-approximation of the processing time, this is done in order to contain the complexity of the problem. With this restricted cache modeling, our problem can be formulated as a standard traveling salesman problem (TSP). However, unlike versions of TSP that are anemable to reasonable approximations, the TSP arising in our case is a non-metric, asymmetric TSP that cannot be approximated within any factor [3]. Hence, we have used the well-known Lin-Kernighan local search heuristic (LKH) [3], which is known to perform well for this class of TSP problems [9].

Apart from the sensor ordering, we also study the impact of code layout on the power consumption in our bio-monitoring application. Clearly, the maximum power savings can be obtained by jointly choosing a code layout *and* a sensor ordering. However, this leads to a combinatorial explosion. Hence we develop a heuristic to choose a near-optimal code layout. The heuristic works on a weighted cache conflict graph whose nodes are procedures, and the edge weights capture cache misses incurred due to instruction cache interference by two given procedures. Based on this cache conflict graph, a code layout is obtained where the procedure causing maximum cache evictions is placed first, and procedures causing lesser cache evictions are placed later. We then find the best sensor ordering for the code layout thus obtained. Our experiments on the fall detection application suggest (a) the impact of sensor ordering on the power consumption far exceeds the impact of code layout, and (b) our heuristic for code layout exploration finds the best possible code layout in terms of power savings.

Organization of this paper: In the next section we describe our fall detection application in detail, followed by our cache modeling technique in Section 3 in conjunction with the exhaustive search for determining the optimal sensor ordering. We then present our restrictive cache

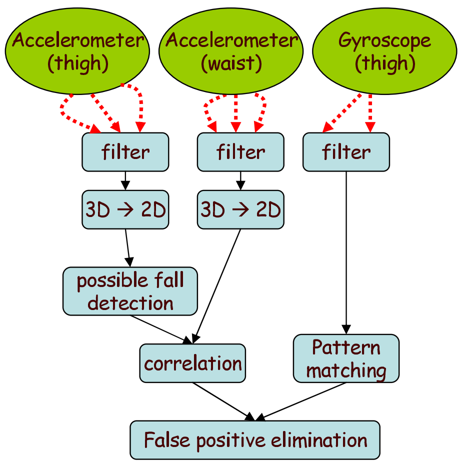


Figure 2: Structure of the fall detection application.

modeling and discuss how it is used along with the LK heuristic. Experimental results are presented in Section 5. Finally, we conclude in Section 6 by discussing some directions for future work.

2 Case-Study Application

Any wearable fall detection system typically employs physical motion sensors such as tri-axial accelerometers and gyroscopes. The fall detection system we examine as a case study consists of one tri-axial (3D) MEMS accelerometer plus one gyroscope on the thigh position and another accelerometer on the waist position. The sensitivity axes of each accelerometer is arranged in lateral, vertical, and antero posterior directions. The gyroscope provides 2D angular (lateral and sagittal) motion information. Overall we have eight streams of sensor signals coming in from the physical motion sensors (lateral, vertical, antero-posterior for each accelerometer and lateral, sagittal for gyroscope) to the gateway device through ZigBee (802.15.4) wireless communication protocol. The sensor signals are sampled at 300 samples per second to detect the onset of falls. The fall detection algorithm runs on the gateway device. In the event that the onset of a fall is detected, the fall safety system is engaged that may attempt to minimize the injury through airbag inflation or prevent the fall through muscle constriction. In any case, the gateway device notifies the incident to the heath care providers or relative of the patient via mobile telephone networks (GPRS, 3G, etc.) or wireless LAN.

The algorithm implementing fall detection first needs to transform the 3D accelerometer data to 2D angular data (lateral and sagittal). Next, it marks an angular motion of the thigh beyond a threshold as a “possible” onset of fall. For each such possible onset of fall, the correlation between thigh and waist angles as well as pattern matching of gyroscope angle (against reference values obtained from a number of actual falls) are used to eliminate false positives. A high-level overview of the functionalities of this application appears in Figure 2.

Let us now examine the code sharing pattern among the program paths exercised by the eight different sensors in this application to intuitively understand the opportunities for instruction cache reuse. Figure 3 shows the different modules (procedures or functions) for a small frag-

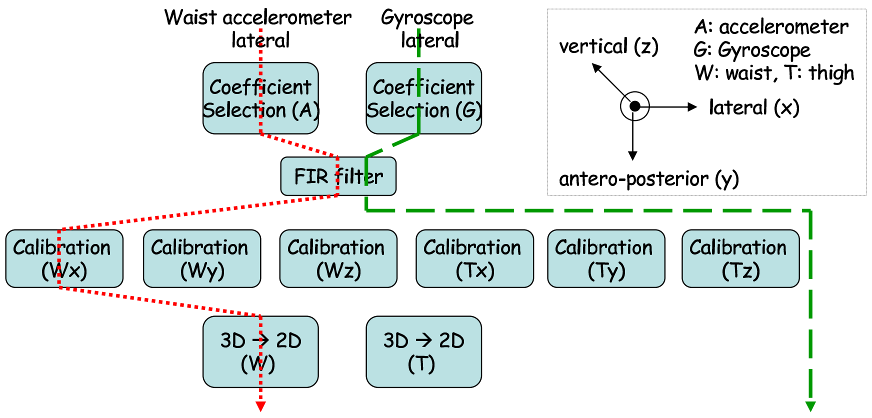


Figure 3: A fragment of the application with different sensor data exercising different paths through the application.

ment of the application code. Each of the sampled sensor data first undergoes discrete wavelet transformation by going through two complimentary filters (a low-pass filter and a high-pass filter). However, the coefficients for the acclelerometer signals and gyroscope signals are quite different. So we have two different coefficient selection modules. Next, the accelerometer signals are calibrated and the calibration process is different for each signal. Finally, each 3D accelerometer signal is converted to 2D angular motion data by going through the appropriate 3D $\rightarrow$ 2D module.

Figure 3 also shows the modules exercised to process the lateral motion from the waist accelerometer say S (left path) and the lateral motion from the gyroscope say S' (right path). Clearly, the processing of these two sensor data shares the FIR filter code. If S' is processed after S, then S' can take advantage of the common code left in the instruction cache by S (such as FIR filter code). Moreover, in this small fragment of the application, it is quite obvious that sampling the accelerometer signals first followed by gyroscope signals will provide maximum opportunities for cache reuse (as the code for coefficient selection may be reused in addition to FIR filter). In general it is non-trivial to manually identify the optimal sampling order of the different sensors for maximal cache re-use, especially when the number of sensors is large and/or the application is complex.

Traditional code optimization approaches are oblivious to the context in which a sensor data is being processed. In other words, the processing of each sensor data is considered in isolation. Thus these code optimization techniques fail to exploit the cache reuse opportunities opened up by code sharing among the processing tasks of the different sensors. In the next section we present a systematic methodology to exploit maximal cache reuse through optimal sampling order of the sensors.

3 Choosing a Sensor Ordering

In this section, we describe our optimization scheme based on micro-architectural modeling, in particular, cache modeling. Further, we determine the order in which the different sensors should be sampled to maximize the *guaranteed cache re-use*. As an example, consider two sensors S and S' — which are processed by programs P and P'. If the two programs are completely disjoint, we do not encounter any instruction cache re-use (owing to the execution of P) while executing P'. However, typically, the processing of different pieces of sensed data share (a lot of) common code. Consequently, there is non-trivial cache re-use from the execution of P while we are executing P'. However, since P and P' are programs with many possible execution traces, the cache re-use (the number of cache hits thus accrued) is not a constant. Depending on the exact value of the sensor data — both P and P' may execute different paths. Executing a trace π_1 of P just prior to a trace π'_1 of P' may produce many additional hits over executing some other trace π_2 of P prior to another trace π'_2 of P'. Therein lies our notion of *guaranteed* cache re-use. The guaranteed cache re-use of a given ordering of sensors $S_1, \ldots, S_k$ stems from the commonality of the processing code for $S_1, \ldots, S_k$ *irrespective of the paths executed* in these processing codes. Note that this involves *static* program analysis (we are analyzing the programs processing the sensor data). The static analysis results are used to find a sensor ordering which maximizes the guaranteed cache re-use.

3.1 Cache Behavior Summarization

We now formally describe our static analysis method for computing cache behavior summary for a given application program. To model cache behavior, we first need the notion of a *cache state*. For simplicity of notation, let us assume a direct-mapped cache; the analysis can be straightforwardly extended for set-associative caches. For a direct-mapped cache with n blocks, a cache state cs is simply a mapping $\{1, \ldots, n\} \rightarrow M \cup \{\perp\}$, where M is the set of code memory blocks being mapped to cache, and $\perp$ indicates the situation where a cache block is empty. As a notational shorthand, we use $cs[i]$ to denote the content of the ith cache block in cache state cs.

Now, let us consider a program processing a particular sensed data. In order to statically summarize the overall cache behavior, we associate program points or control locations in the program with *sets of cache states*. We develop and use two quantities: Reaching Cache States (RCS) and Live Cache States (LCS).

Definition 1 (Reaching Cache States) *For a program point p in a program $Prog$, the set of reaching cache states $RCS(p)$ is defined as the set of cache states with which p can be reached (via any incoming path to p in $Prog$).*

Definition 2 (Live Cache States) *For a program point p in a program $Prog$, the set of live cache states $LCS(p)$ is the set of possible first references to cache blocks via any outgoing path from p in $Prog$.*

Given any program point p in program $Prog$, the quantities $LCS(p)$ and $RCS(p)$ are computed by exploring the paths to/from p in the control flow graph of $Prog$. This is done efficiently (without path enumeration) by (i) associating each program point with a LCS/RCS, (ii) defining the RCS of a program point using the RCS of its predecessors, and (iii) defining the LCS of a program point using the LCS of its successors. As a program contains loops, the above will produce a set of recursive equations on LCS/RCS that needs to be solved iteratively. Assuming empty cache at the beginning of the program, we can iteratively solve the recursive equations for LCS and RCS separately. This is done, until the LCS and RCS estimates at each program point is stable – that is, until the iterative computation reaches a fixed-point.

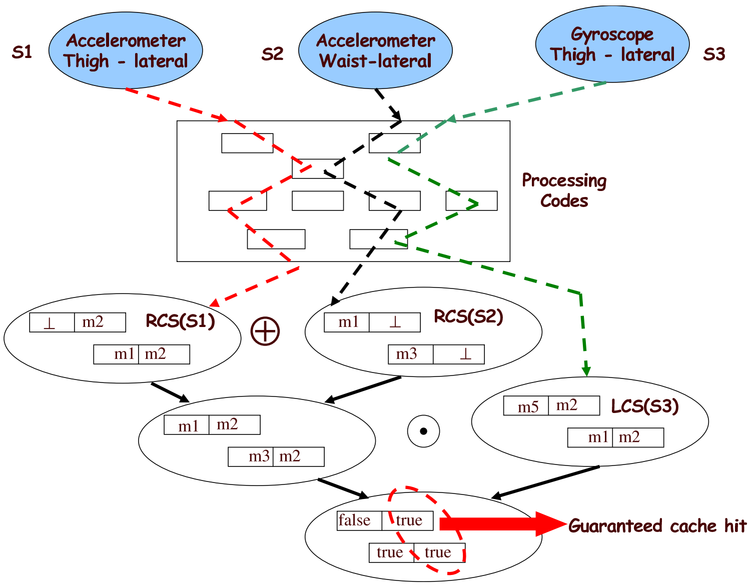

S1
Accelerometer Thigh - lateral
S2
Accelerometer Waist-lateral
Gyroscope Thigh - lateral
S3
Processing Codes
RCS(S1)
RCS(S2)
LCS(S3)
Guaranteed cache hit

Figure 4: Computing Guaranteed Cache Re-use.

The resultant RCS estimate for the exit point of the program is denoted as $RCS(Prog)$; these are the possible cache states at the end of the program. Similarly, the LCS estimate at the entry point of the program is denoted as $LCS(Prog)$; these are the possible first references to cache blocks during the program's execution. Given a program $Prog_i$ processing a particular sensed data S_i, the quantities $RCS(Prog_i)$ and $LCS(Prog_i)$ form the *summary of the cache behavior* for $Prog_i$. Details of LCS and RCS computation for a program appear in [15].

3.2 Composition of Cache Summaries

Consider the processing of various sensed data $S_1, \ldots, S_k$ where the processing of S_i is done by application $Prog_i$. In reality, for $i \neq j$, applications $Prog_i$ and $Prog_j$ are not disjoint — they share non-trivial chunks of code. In the preceding, we have shown the cache behavior summarization of each individual application. Now we compose these cache summaries to tightly estimate the cache re-use from a given ordering of processing. When we say a given ordering $S_1, \ldots, S_k$ we mean that (i) data from these sensors are repeatedly obtained over many "rounds", and (ii) in each round, the sensed data in that round are processed by executing $Prog_1$, followed by $Prog_2$, ..., followed by $Prog_k$.

So, we need to estimate the minimum guaranteed cache re-use resulting from the execution of $Prog_1, \ldots, Prog_k$. At this stage, we are determining the cache behavior *across* applications. The cache behavior within an application is already summarized by the LCS/RCS quantities. The cache behavior across applications can be intuitively captured by queries like the following.

- Given the execution of $Prog_1, \ldots, Prog_i$, what is the guaranteed cache re-use when we execute $Prog_{i+1}$?

To answer the above query, let us try to estimate the set of possible cache states after the execution of $Prog_1, \ldots, Prog_i$. For this purpose, we define a simple operation over cache blocks. Note that a cache block's content is drawn from $M \cup \{\perp\}$ where M is the set of memory blocks and $\perp$ is a symbol denoting empty cache block. We define:

$$m \oplus m' \overset{def}{=} \begin{cases} m' & \text{if } m' \neq \perp \\ m & \text{otherwise} \end{cases}$$

Thus $m \oplus m'$ is m' (the later content) unless it is empty. As a cache state is essentially a vector of cache block contents, the above operation can be lifted to cache states by applying the operation point-wise to individual cache blocks.

$$(cs \oplus cs')[i] \overset{def}{=} cs[i] \oplus cs'[i] \;\; 1 \leq i \leq n$$

The operation can now be further lifted to *sets* of cache states CS, CS' as $CS \oplus CS' \overset{def}{=} \{cs \oplus cs' \mid cs \in CS, cs' \in CS'\}$. Using this operator, the set of possible cache states after the execution of $Prog_1, Prog_2$ is simply $RCS(Prog_1) \oplus RCS(Prog_2)$. That is, the set of cache states is drawn from the execution of $Prog_2$ (the application last executed), but for empty cache blocks their content is derived from $RCS(Prog_1)$. In a similar way, the set of cache states after the processing of sensed data $S_1, \ldots, S_i$ (*i.e.*, after executing $Prog_1, Prog_2, \ldots, Prog_i$) is

$$Reach_{1\ldots i} = RCS(Prog_1) \oplus RCS(Prog_2) \oplus \ldots \oplus RCS(Prog_i)$$

Given the above set of cache states after the execution of $Prog_1$, $\ldots, Prog_i$, what is the minimum number of guaranteed cache hits when $Prog_{i+1}$ is executed? To answer this, we need to see how the cache states at the end of $Prog_1, Prog_2, \ldots, Prog_i$ can help (in terms of achieving cache hits) the possible first references to cache blocks in $Prog_{i+1}$. Recall, the possible first references to cache blocks in an application is summarized by the LCS of the application. So, the hit/miss scenarios encountered in $Prog_{i+1}$ due to the prior execution of $Prog_1, \ldots, Prog_i$ is

$$Reach_{1\ldots i} \odot LCS(Prog_{i+1})$$

where $Reach_{1\ldots i}$ is as defined in the preceding (using RCS quantities) and the operator $\odot$ defines cache state equality as follows. Given cache states cs, cs' we get a vector of boolean values as follows: $(cs \odot cs')[i] \overset{def}{=} (cs[i] == cs'[i])$. Thus, $cs \odot cs'$ checks whether the cache states cs, cs' are equal; for each cache block that is equal in content it stores the boolean value *true* (otherwise *false*). We can lift the $\odot$ operation over *sets* of cache states CS, CS' in the usual way:

$$CS \odot CS' \overset{def}{=} \{cs \odot cs' \mid cs \in CS, cs' \in CS'\}$$

Hence, the set $Reach_{1\ldots i} \odot LCS(Prog_{i+1})$ summarizes all the hit/miss scenarios for the first cache block accesses in $Prog_{i+1}$ owing to the prior execution of $Prog_1, \ldots, Prog_i$. The minimum guaranteed number of cache hits is given by:

$$min_{h \in Reach_{1\ldots i} \odot LCS(Prog_{i+1})} \; |h|$$

where h is a vector of boolean values in $Reach_{1\ldots i} \odot$ $LCS(Prog_{i+1})$ and $|h|$ is the number of occurrence of *true* in the boolean vector h.

A schematic explanation of the guaranteed cache re-use computation appears in Figure 4. Here we show the processing of three sensors $S1, S2, S3$ and the guaranteed cache re-use encountered while processing $S3$ assuming prior processing of $S1, S2$. We have shown a direct-mapped cache with only two blocks for simplicity of explanation. Effect of the prior processing of $S1, S2$ is captured by $RCS(S1) \oplus RCS(S2)$. This set is computed by applying the $\oplus$ operator pairwise to the members of $RCS(S1)$ and $RCS(S2)$. Thus $\langle \perp, m2 \rangle \oplus \langle m1, \perp \rangle = \langle m1, m2 \rangle$. Once $RCS(S1) \oplus RCS(S2)$ has been computed, we check for cache state equality with $LCS(S3)$, the possible first references to cache blocks in the processing of $S3$. Again, $(RCS(S1) \oplus RCS(S2)) \odot LCS(S3)$ is computed by applying the $\odot$ operation pairwise. Thus, by applying $\odot$ to $\langle m1, m2 \rangle$ (a cache state in $RCS(S1) \oplus RCS(S2)$) and $\langle m5, m2 \rangle$ (a cache state in $LCS(S3)$), we get the boolean vector $\langle false, true \rangle$ signifying a cache hit in the second cache block. By analyzing all such possible hit/miss scenarios (in this simple example there are only two of them as shown in Figure 4), we calculate the number of guaranteed cache hits during the processing of $S3$ due to the prior processing of $S1, S2$.

In the preceding discussion, we have detailed the cache behavior summarization for (a) any one application, and (b) across many applications, provided an ordering for their execution is given. We now describe how these cache behavior summaries can be exploited to produce an "optimal" ordering of the applications' execution. The generated ordering is optimal in the sense that it maximizes cache re-use.

3.3 Determining Optimal Composition Order

For an application with k sensors, in each ordering $S_1, \ldots, S_k$, the cache re-use can be calculated as:

$$\sum_{j=1}^{k-1} (min_{h \in Reach_{1\ldots j} \odot LCS(Prog_{j+1})} |h|)$$

We can compute the cache re-use for all $k!$ different orderings and find the "optimal" ordering with maximum cache re-use. However, for larger number of sensors, such an exhaustive search becomes infeasible. We propose to convert our problem of finding the "optimal" ordering into the well-known Traveling salesman problem (TSP). Given k sensed data for processing, $(S_1, \ldots, S_k)$, we define a complete graph G where each vertex represents a sensor. The weight $w(S_i \to S_j)$ of the edge $S_i \to S_j$ represents the guaranteed number of cache hits encountered while processing S_j owing to the processing of S_i immediately prior to it. Thus

$$w(S_i \to S_j) \overset{def}{=} min_{h \in RCS(Prog_i) \odot LCS(Prog_j)} |h|$$

Thus, we are only considering the application program $Prog_i$ that executed immediately before $Prog_j$. We are *not* considering in the execution history which application program(s) executed prior to $Prog_i$. Thus, the guaranteed cache re-use estimated by us will be a (safe) underestimate of the actual cache re-use. However, this under-estimation is relatively small, and ignoring it still produces near-optimal results. Finally, note that $w(S_i \to S_j) \neq w(S_j \to S_i)$ as the cache re-use of S_i from S_j could be different from the re-use of S_j from S_i.

For a small fragment of our fall detection application, Table 1 shows the guaranteed number of cache hits for the task associated with sensor S_j if the immediately preceding sensor whose data was processed is S_i. The guaranteed cache hits for the optimal sampling order are indicated in bold face. It may be noted that even for such relatively few sensors, determining this order optimally in an ad hoc trial-and-error fashion might not be possible. Therefore, we formulate it as a TSP.

		S_j							
		s1	s2	s3	s4	s5	s6	s7	s8
S_i processing order: (S_i, S_j)	s1	-	45	34	36	**45**	45	17	18
	s2	29	-	24	**25**	32	34	16	19
	s3	11	16	-	16	14	16	13	11
	s4	**39**	39	28	-	37	38	14	15
	s5	43	48	37	39	-	**48**	20	21
	s6	21	26	**24**	17	24	-	8	18
	s7	33	**38**	28	29	36	38	-	23
	s8	25	30	23	22	28	30	**21**	-
Best ordering: 240 cache hits for $(S_8, S_7, S_2, S_4, S_1, S_5, S_6, S_3)$									

Table 1: Guaranteed cache reuse from previous sensor task.

TSP is defined as finding a hamiltonian cycle (a tour) of a graph G with minimum cost. To solve our problem as a TSP, we need to make the following modifications. (1) TSP finds hamiltonian cycles where each node is visited exactly once. However, our problem of searching for the best ordering is to find an acyclic path in the graph, where each node is visited exactly once. Hence, we add a dummy node S_{k+1} to G. Edges $S_{k+1} \rightarrow S_i$ and $S_i \rightarrow S_{k+1}$ are set with weight 0 for all nodes S_i ($1 \leq i \leq k$). The dummy node tells us where to break the cyclic tour to generate the optimal linear ordering. (2) While TSP returns a tour with the minimum cost, we need to find a path with maximum cost instead (maximum cache re-use). Hence, for each edge, weight $w(S_i, S_j)$ is modified to $Const - w(S_i, S_j)$ for all edges, where $Const$ is a large constant bigger than any edge weight of G.

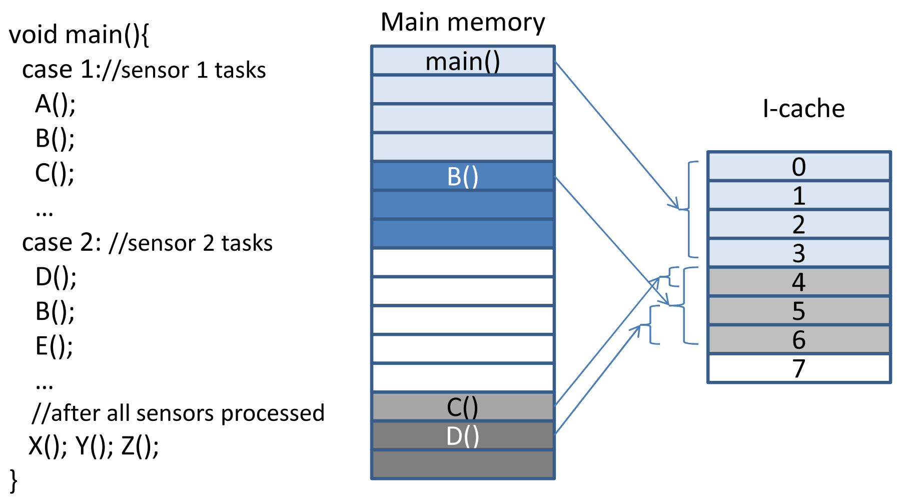
After these two modifications above, we apply the well-known Lin-Kernighan local search heuristic (LKH) [9] to find the tour of G with minimum cost. LKH is known to perform well for non-metric, asymmetric TSP; it produces near-optimal results for our experiments as well.

4 Code Layout Exploration

In this section, we develop a heuristic to find a near-optimal code layout. Since jointly optimizing for code layout and sensor ordering leads to combinatorial explosion — we find a code layout and then find the best sensor ordering for the code layout thus found.

Let us step back and find out why at all code layout is an issue in terms of cache optimization of a given application. For a BAN application and a particular hardware configuration, the instruction cache utilization depends on not only the ordering of sensor tasks been processed, but also the application's code layout in the memory. Consider two sensor tasks S_1 and S_2, which execute the function sequences (A, B, C) and (D, B, E) respectively. Suppose instructions in functions A, B, and D are all mapped to the same instruction cache blocks (assuming a direct mapped cache). In this case, neither sensor ordering (S_1, S_2) nor (S_2, S_1) gives any cache reuse due to the second invocation of function B, since instructions of function A or D will replace B's cached instructions after its first invocation.

Cache-aware code placement is a well studied compiler optimization to increase instruction cache utilization, and hence improve the overall performance of an application. [20] performs

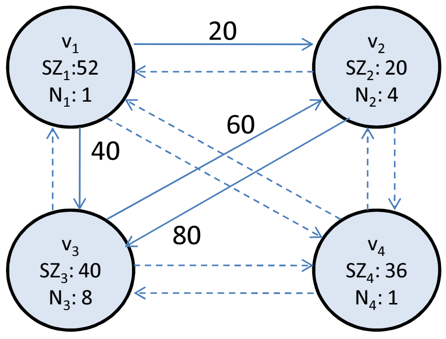


```
void main(){
 case 1://sensor 1 tasks
  A();
  B();
  C();
  ...
 case 2: //sensor 2 tasks
  D();
  B();
  E();
  ...
  //after all sensors processed
 X(); Y(); Z();
}
```

Figure 5: Call frequency based procedure placement.

code placement based on program execution profile information, which reduces the cache miss rates for average case execution time (in contrast to the worst case analysis in our problem setting). In [20], the code placement is done at basic block level, so that additional branching statements need to be inserted into original code, which increases the code size. The work of [14] proposes an iterative approach targeting for WCET reduction via procedure positioning. Based on the call graph and call frequencies information of the WCET path for current program layout, the analysis tries to find the next program layout with a smaller WCET by placing procedures with high call frequencies contiguously in memory (so that they do not conflict with each other in the cache). However, call frequency based procedure placement techniques (e.g., [19], [14]) do not fit for the specific BAN application we are trying to optimize in this paper. Figure 5 shows an example of procedure placement using call frequency based approach for a direct mapped instruction cache. Suppose procedure $B()$ is a common task invoked by $main()$ for processing each sensor task. Thus, the highest call frequency in the call graph is between $main()$ and $B()$. However, placing $main()$ and $B()$ contiguously in the code memory does not give a good instruction cache utilization, because procedures called by $main()$ in between two consecutive invocations of $B()$ (e.g., $C()$ and $D()$ in Figure 5) can be mapped to the same cache blocks as $B()$ and cause subsequent invocations of $B()$ to incur cache misses. In other words, for the particular sensor processing ordering (S_1, S_2), even though procedure $C()$ and $D()$ are not invoked by $B()$, their relevant positions to $B()$ must be considered in order to improve the cache behavior.

Our problem of cache performance optimization of BAN application is more complex than previously studied code placement-based optimizations. The maximum guaranteed cache reuse in one execution of the application (which leads to a tight WCET result) depends on both the sensor task processing ordering and application code layout. The two factors are correlated to each other, which makes the joint optimization a difficult problem. For the application consisting of 42 procedures and 8 different sensor tasks (to be processed in any order), the design space contains $(42! \times 8!)$ different combinations of possible procedure layout and sensor ordering, which makes the full-space searching for an optimal combination infeasible. In this paper,



(a) I-cache conflict graph

Step 1: $p_3 \rightarrow$[c0-c39]
Step 2: $p_3 \rightarrow$[c0-c39]
$p_2 \rightarrow$[c40-c59]
Step 3: $p_3 \rightarrow$[c0-c39]
$p_2 \rightarrow$[c40-c59]
$p_4 \rightarrow$[c60-c63],[c0-c31]
Step 4: $p_3 \rightarrow$[c0-c39]
$p_2 \rightarrow$[c40-c59]
$p_4 \rightarrow$[c60-c63],[c0-c31]
$p_1 \rightarrow$[c32-c63],[c0-c19]
Eviction on p_3 due to p_1: 28
Eviction on p_2 due to p_1: 20
Total cache evictions: 48

(b) Our "good" layout

Step 1: $p_3 \rightarrow$[c0-c39]
Step 2: $p_3 \rightarrow$[c0-c39]
$p_4 \rightarrow$[c40-c63],[c0-c11]
Step 3: $p_3 \rightarrow$[c0-c39]
$p_4 \rightarrow$[c40-c63],[c0-c11]
$p_1 \rightarrow$[c12-c63]
Step 4: $p_3 \rightarrow$[c0-c39]
$p_4 \rightarrow$[c40-c63],[c0-c11]
$p_1 \rightarrow$[c12-c63]
$p_2 \rightarrow$[c0-c19]
Eviction on p_3 due to p_1: 28
Eviction on p_2 due to p_1: 8
Eviction on p_3 due to p_2: 80
Eviction on p_2 due to p_3: 60
Total cache evictions: 176

(c) Another worse layout

Figure 6: Example I-cache conflict graph and possible code layouts.

we propose a simple heuristic to find a near-optimal layout, then use the technique described in Section 3.3 to find the optimal sensor ordering for this layout.

4.1 Cache Conflict Graph

Our greedy heuristic approach to generate a near-optimal layout is based on an instruction cache conflict graph $G = \langle V, E \rangle$. A node $v \in V$ represents a procedure in the application, and is associated with its (maximum) invocation count N_v in one complete execution of the application. A directed edge $e_{i \rightarrow j} \in E$ from node v_i to v_j is weighted with number of maximum possible additional cache misses incurred by v_j due to possible calls to v_i in between consecutive invocations of v_j. Let SZ_v denotes the size (number of instructions) of procedure v, the weight of edge $e_{i \rightarrow j}$ can be calculated as:

$$W_{e_{i \rightarrow j}} = min\{N_{v_i}, (N_{v_j} - 1)\} \times min\{SZ_{v_i}, SZ_{v_j}\}$$

since the number of time v_j's cached instructions get replaced by v_i's instruction depends on the minimum between N_{v_i} and $N_{v_j} - 1$ (due to the cold miss of v_j's first invocation); and for each such interference, number of v_j's instructions replaced by v_i is bounded by the minimum size of procedures v_i and v_j. Clearly, for procedures only invoked once during the application's execution, all incoming edges are weighted with 0. Furthermore, for pairs of procedures that are guaranteed to have mutual exclusive lifetime, the cache conflicts between them are set to be 0. For example in the code shown in Figure 5, procedures $X()$, $Y()$ and $Z()$ are invoked only after execution of all individual sensor processing tasks. Regardless of the sensor ordering, these procedures will not cause any cache conflicts with sensor processing tasks (e.g., $B()$). Figure 6(a) shows an example of a cache conflict graph with four nodes (procedures). Edges are labeled with the conflict values between two nodes, and dashed edges are with weight 0. For instance, v_4's lifetime is assumed to be mutual exclusive with the remaining three procedures, thus all edges connecting from (to) v_4 are assigned a weight 0.

Algorithm 1 Greedy algorithm to generate a near-optimal layout

```
1: INPUT: A set $P$ of all procedures (excluding $main()$) in the application program
2: OUTPUT: A optimized program layout (procedure ordering) $L$
3:
4: $L = \emptyset$
5: $G$ = buildConflictGraph($P$)
6: initProc = maxConflict($G$)
7: append($L$, $initProc$)
8: repeat
9:     $nextProc = \emptyset$;  $minConflict = +\infty$
10:     for (each unplaced procedure $p_i \in P$) do
11:         tmpConflict = computeConflict($L$,$p_i$,$G$)
12:         if ($tmpConflict < minConflict$) then
13:             $nextProc = p_i$
14:         else
15:             if ($tmpConflict == minConflict$) then
16:                 $nextProc$ = chooseProc($p_i$,$nextProc$,$G$)
17:             end if
18:         end if
19:     end for
20:     append($L$, $nextProc$)
21: until (all procedures $p \in P$ are placed into $L$)
22: append($L$, $main()$)
```

4.2 Greedy Algorithm to Produce a Layout

The greedy algorithm to generate a near-optimal code layout (with the goal of reducing misses in the instruction cache) is shown in Algorithm 1. The above-mentioned instruction cache conflict graph G is initially built based on the information of procedures and corresponding lifetimes (Line 5). The procedure in G with maximum aggregated sum of *incoming* edges' weight is chosen (Line 6) and appended to the program layout L (Line 7). Subsequently, the remaining procedures are appended to the end of L one at a time (Line 8 - 21). In each iteration, each remaining procedure is tested on the maximum cache evictions on the existing placed procedures in L if it is placed at the end of L (Line 11). Given procedure p and existing procedure p' in the layout L, the maximum cache eviction on p' because of appending p is the number of interferences between p and p' multiplied by the actual layout conflicts by taking the layouts of p and p' in L into account. The procedure results in minimum cache eviction is chosen to append to L (Line 20). If multiple remaining procedures cause the same number of cache conflicts, a heuristic is applied to choose the procedure with maximum aggregated sum of all its *outgoing* edges' weights (Line 16).

We illustrate the algorithm using the cache conflict graph example shown in Figure 6(a). We assume a direct mapped instruction cache with 64 cache blocks $[c0, ...c63]$. Without loss of generality, we also assume the starting memory address of the first procedure in the generated layout is mapped to cache block $c0$. Figure 6(b) shows the steps and final output layout from our greedy algorithm. Procedure p_3 is first added into the memory layout and mapped to cache blocks $[c0 - c39]$. Placing either p_2 or p_4 next to p_3 will result in 0 cache eviction. Due to our heuristic at Line 16, p_2 is placed in the second step, which is mapped to cache blocks $[c40 - c59]$. p_4 is placed after p_2, which cause no cache eviction on p_3 and p_2 according to the cache conflict graph. Finally, p_1 is placed after p_4, and mapped to cache blocks $[c32 - c63]$ and $[c0 - c19]$, which causes 28 cache evictions on p_3 (sharing cache blocks $[c0 - c19]$ and $[c32 - c39]$) and 20 evictions on p_2 (sharing cache blocks $[c40 - c59]$). On the other hand, without the heuristic at

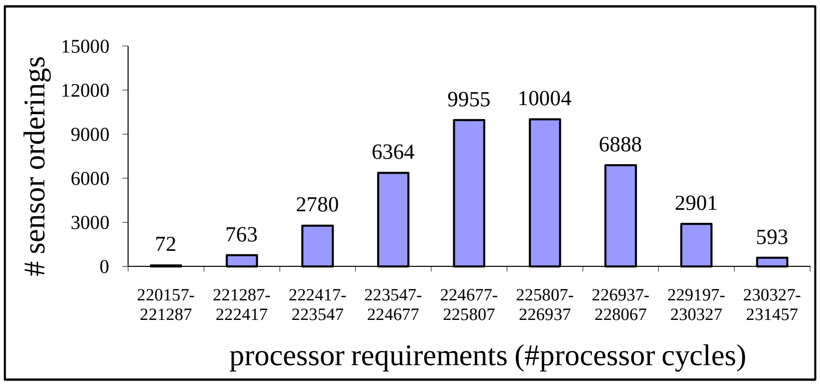


Figure 7: Execution requirements for different sensor orderings.

Line 16, if any procedure with minimum cache evictions with p_3 is chosen to be placed at step 2 (e.g., p_4), the generated layout may result in a worse cache behavior (as shown in Figure 6(c)).

Finally, recall that given the code layout produced by Algorithm 1, we will employ the methods presented in Section 3 to find the best sensor ordering for this layout.

5 Experimental Results

We have conducted two different classes of experiments. The first using the full-fledged fall detection application that was described in Section 2. Here, our experiments illustrate the utility of our proposed cache modeling in tightly estimating the gateway processor's minimum clock frequency and reducing its power dissipation. Our second class of experiments are based on synthetic data and a larger number of sensors (15 - 20 in number). Here, our main goal is to illustrate the minimal loss in accuracy as a result of the restrictive cache modeling.

Experimental Setup for Fall Detection Application Recall that our application has three sensor inputs from the accelerometer and two from the gyroscope attached to the thigh, and three sensor inputs from the accelerometer attached to the waist (Figure 2). For the gateway, we have assumed a light-weight processor with single-issue in-order pipeline, 1 KB direct-mapped instruction cache (128 cache sets, 8 bytes block size) and 100 cycles cache miss penalty. We have also used a "natural" code layout that resulted from compiling the original C code of the application using SimpleScalar GCC compiler [1]. For all power estimates, we used Wattch [2] along with the SimpleScalar instruction set simulator.

Power Savings due to Sensor Ordering Our experimental results show that for the original natural code layout, the number of processor cycles required for processing all the data from one *round* of sampling (i.e., one data sample from each of the eight sensors) is equal to 248,657 cycles, when no inter-application (i.e., processing code for different sensors) cache reuse is modeled. In other words, the instruction cache is assumed to be empty before the application code for each sensor starts executing. With inter-application cache modeling, an optimal ordering of the sensors results in 220,157 cycles, whereas the worst-case ordering results in

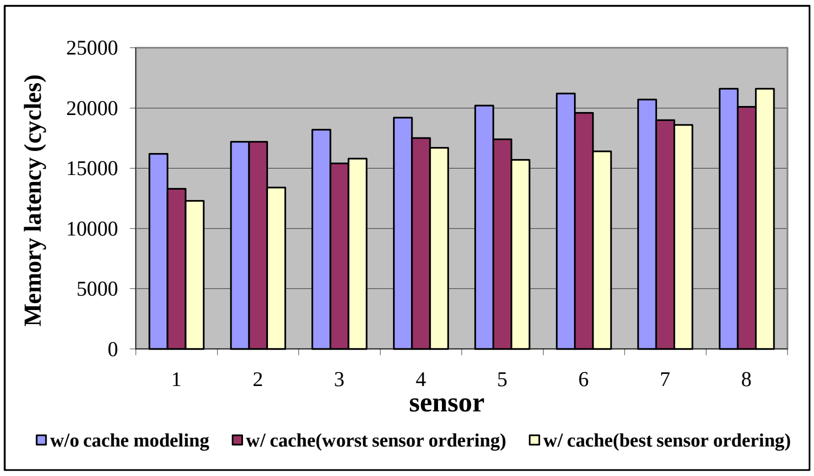


Figure 8: Memory latency reduction for sensor data processing tasks.

231,457 cycles. Note that the chances of an arbitrary ordering being close to the optimal is fairly low. This is illustrated in Figure 7 where the range of execution requirements (i.e., number of processor cycles) for one round of processing has been partitioned into 9 equal-sized bins (horizontal axis). Each bar in this figure represents the number of sensor orderings that result in the execution requirement corresponding to the associated bin. The bin with the lowest execution requirement (first bin) contains only 72 of the $8!$ different possible sensor orderings. This illustrates the need for a systematic approach to optimize the sensor sampling order. The importance of accurate cache modeling is illustrated in Figure 8, which shows the memory latencies (in number of processor cycles) for the tasks associated with the different sensors, with and without cache modeling. It also shows the reduction in memory latency with optimal ordering (except for sensor 8, which being the first sensor in this ordering cannot exploit any cache reuse). With a sampling rate of 300 samples/sec and no inter-task cache modeling, the processor is estimated to be clocked at 74.6 MHz. With the same sampling rate, but with inter-task cache modeling the estimated minimum clock frequencies drop to 69.5 MHz and 66.1 MHz for the worst and the best sensor orderings. Hence, without accurately modeling the instruction cache and ignoring the sampling ordering of the sensors can lead to running the processor at a 13% higher clock frequency. We simulate the applications over 100 rounds using Wattch and compute the average power consumption. As a result, by running the original application (natural code layout) with the optimal sensor processing ordering, the average power consumption can be reduced by a factor of 19.88%.

Power Savings due to Code Layout Our application contains 42 procedures, which makes it infeasible to exhaustively search all possible code layouts and their corresponding best-case sensor orderings in order to obtain the optimal cache optimization solution ($42! \times 8!$ combinations). We have randomly generated 100,000 different layouts and calculated WCET of the application with best-case sensor ordering for each of these layouts. Figure 9 shows the WCET distribution of the 100,000 layouts. The “best” layout among them gives WCET of 214,157 with its best-case sensor ordering, which is 2.7% smaller than the WCET of natural code layout. We

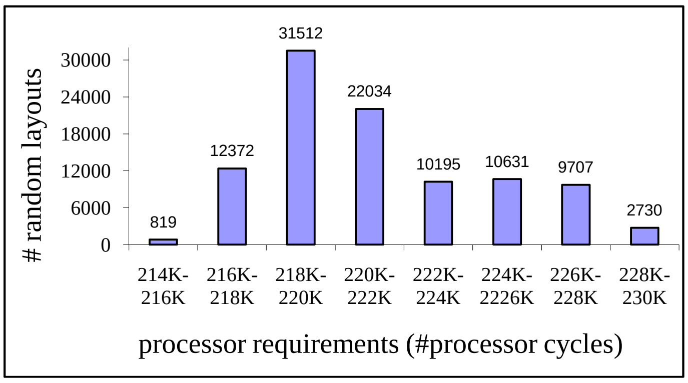


Figure 9: WCET with best-case sensor ordering for 100,000 random code layouts of the application.

have also applied the heuristic algorithm as described in Section 4.1 to produce a code layout. WCET of our generated layout with best-case sensor ordering is 215,657 (2.3% smaller than the WCET of natural code layout). Based on the required sampling rate of 300 samples/sec, minimum clock frequency to execute our application with this "good" layout is 64.7 MHz, leading to a total 22.93% reduction on average power consumption (comparing with running the processor at 74.6 MHz for natural code layout without sensor ordering optimization). Choice of code layout only achieves additional 3.05%(22.93%-19.88%) average power consumption reduction comparing to natural layout.

Experiments with Synthetic Datasets For large number of sensors, exhaustively enumerating all possible sensor orderings is not feasible. Hence, we rely on the TSP formulation of the problem (as described in Section 3.3) to identify the optimal ordering. The question remains about the quality of the solutions returned by this TSP formulation. There are two sources of approximations in this formulation: (1) it maintains a limited cache history, and (2) it applies the LK heuristic to solve the TSP problem. While the sub-optimality due to the LK heuristic is a well studied issue (see [3]), the effect of the first approximation is unknown. First, we observe that for our case study, there exists up to 5% additional guaranteed cache reuse beyond the immediate predecessor sensor. But the TSP formulation returns identical sensor orderings as the exhaustive search.

To confirm the generality of this result, we modeled cache reuse beyond just the immediate predecessors in an experiment with synthetic data. We assigned random weights (within appropriate constraints) for cache re-use between two sensor nodes as well as three sensors sequences. However, as cache sizes in gateway devices are quite small, we do not model cache reuse beyond two sensor nodes. We compare the quality of the solutions (in terms of guaranteed cache hits) returned by TSP formulation and exhaustive search for 12 sensor nodes. The TSP formulation returns optimal or near-optimal solutions (with more than 95% accuracy) in all the cases. Moreover, we observed that exhaustive search is scalable only up to 15 sensors. But our TSP formulation in conjunction with the LK heuristic returns an ordering within few seconds for hundreds of sensors.

6 Concluding Remarks

In this paper we have proposed a methodical cache-aware optimization technique for wireless Body area Networks (BANs). Our results — with a real-life fall detection application for the elderly — show that appropriate modeling of the gateway processor's cache, coupled with carefully determining the order in which the sensors are sampled can lead to significant energy savings. Moreover, we also observed that near-optimal code placement strategies in conjunction with an appropriate sensor ordering further improves execution time and energy consumption estimates of our biomonitoring application.

7 Acknowledgments

This work is supported by A*STAR SERC project R-252-000-258-305. We would like to thank Francis Eng Hock Tay and Nyan Myo Naing for sharing the fall detection application with us.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[3] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, 2006.

[4] R. W. DeVaul et al. MIThril 2003: Applications and architecture. In *International Symposium on Wearable Computers*, 2003.

[5] J. Edmison et al. E-Textile based automatic activity diary for medical annotation and analysis. In *International Workshop on Wearable and Implantable Body Sensor Networks*, 2006.

[6] I. Al Khatib *et al.* A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: Architectural design space exploration. In *DAC*, 2006.

[7] E. Farella et al. A wireless body area sensor network for posture detection. In *IEEE Symposium on Computers and Communications*, 2006.

[8] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM Transactions on Design Automation of Electronic Systems*, 9(4), 2004.

[9] P. Gupta, A. B. Kahng, and S. Mantik. Routing-aware scan chain ordering. *TODAES*, 10(3), 2005.

[10] R. Jafari et al. Adaptive and fault tolerant medical vest for life-critical medical monitoring. In *ACM Symposium on Applied Computing*, 2005.

[11] R. Jafari et al. Wireless sensor networks for health monitoring. In *International Conference on Mobile and Ubiquitous Systems*, 2005.

[12] J.-C. Kao and R. Marculescu. On optimization of e-textile systems using redundancy and energy-aware routing. *IEEE Trans. on Computers*, 55(6), 2006.

[13] I. Al Khatib et al. Performance analysis and design space exploration for high-end biomedical applications: Challenges and solutions. In *CODES+ISSS*, 2007.

[14] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations.

[15] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[17] P. R. Panda, N. D. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *TODAES*, 2(4), 1997.

[18] S. Park, K. Mackenzie, and S. Jayaraman. The wearable motherboard: A framework for personalized mobile information processing (PMIP). In *DAC*, 2002.

[19] K. Pettis and R.C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6), 1990.

[20] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4), 1997.