Middleware Design Optimization of Wireless Protocols Based on the Exploitation of Dynamic Input Patterns

Stylianos Mamagkakis
IMEC vzw.
3001 Heverlee, Belgium
mamagka@imec.be

Dimitrios Soudris
VLSI Center-Democritus Uni.
67100 Xanthi, Greece
dsoudris@ee.duth.gr

Francky Catthoor
IMEC vzw.
3001 Heverlee, Belgium
catthoor@imec.be
also Prof. at ESAT/K.U.Leuven

Abstract

Today, wireless networks are moving big amounts of data between mobile devices, which have to work in an ubiquitous computing environment, which perpetually changes at run-time (i.e., nodes log on and off, varied user activity, etc.). These changes introduce problems that can not be fully analyzed at design-time and require dynamic (run-time) solutions. These solutions are implemented with the use of run-time resource management at the middleware level for a wide variety of embedded systems. In this paper, we motivate and propose the characterization of the dynamic inputs of wireless protocols (e.g., input to the IEEE 802.11b protocol coming from IPv4 data fragmentation). Thus, through statistical analysis we derive patterns that will guide our optimization process of the middleware for run-time resource management design. We assess the effectiveness of our approach with inputs of 18 real life case studies of wireless networks. Finally, we show up to 81.97% increase in the performance of the proposed design solution compared to the state-of-the-art solutions, without compromising memory footprint or energy consumption.

1 Introduction

Today the most popular implementation of the Network layer in mobile devices implementing wireless networks is the Internet Protocol (IP from now on). Its instantiations in embedded systems include the IPv4 [1] and the IPv6 [2] protocols. At this level data fragmentation is performed at run-time (also known as packetization). More specifically, bigger datagrams/packets that come from the higher layer (e.g., TCP, UDP, etc.) have to be broken down to smaller pieces of data, in order to fit to the Maximum Transmission Unit (MTU from now on) of the lower layer (e.g., Ethernet, WiFi, etc.). These smaller packets usually do not have the same size as the MTU. Therefore, this data fragmentation results in packets with many different sizes during run-time. These variable-sized packets constitute the dynamic input of the wireless protocol and the Dynamic Memory (DM from now on) allocator, which is responsible for their allocation to the physical memory of the hardware system [16]. The physical memory is usually a SRAM on-chip memory module or an off-chip SDRAM memory module of an embedded system. The DM allocator normally resides on top of the basic Operating System (OS) services. In that sense, it is part of the middleware and it is called by the wireless protocol according to the dynamic input.

The DM allocator itself is a very complex algorithm and the standard DM allocation solutions are activated with the standardized malloc/free functions in C and the new/delete operators in C++. The standardized DM allocators, which try to handle memory fragmentation of general purpose computers with 'one-size-fits-all' solutions, fail to handle efficiently the type of memory fragmentation produced by the specialized dynamic inputs produced by the Network layer, which produce a very specific pattern of fragmentation. This pattern is a bimodal distribution of allocated sizes. This failure results in mediocre performance and increased memory fragmentation. Due to the latter, ultimately also the memory size limit can be reached with even more disastrous consequences. Therefore, specialized DM allocators are needed [5] [4] to achieve better results. Note that they are still realized in the middleware as a thin layer below the OS and usually not in the hardware of the platform.

More specifically, we propose in this paper a specialized DM allocator design that increases performance of memory allocation and de/allocation up to 81%. The proposed, specialized design exploits inherent characteristics of the dynamic inputs of the wireless protocols (which are produced by the data fragmentation at the IP layer). These characteristics set the final memory fragmentation outlook that the DM allocator must deal with. The major contribution of our work is the speed optimization of the memory allocation subsystem at the middleware level, without increasing the final energy consumption and memory footprint, compared to the standardized state-of-the-art solutions. The remainder of the paper is organized as follows. In Section 2, we describe some related work. In Section 3, we analyze the dynamic inputs. In Section 4, we enumerate the relevant de-fragmentation techniques in DM allocator design. Also, we explain our proposed approach of a specialized DM allocator design. In Section 5 we present the simulation results of our proposed DM allocator and compare it with state-of-the-art solutions. Finally, in Sect. 6 we draw our conclusions.

2 Related Work

Software optimizations of wireless network algorithms have been extensively studied in the related literature. [6,11] are good overviews about the vast range of proposed techniques to improve the performance and energy consumption of wireless networks. These improvements are achieved with the transformation of the algorithms at the Network layer. Hardware optimizations of wireless devices have also been extensively studied in the related literature. [15] is a good overview about the vast range of proposed hardware solutions to improve the efficiency of data manipulation at the Network layer. These solutions come in the form of specialized hardware network protocol processors or other specialized hardware memory allocation modules. On the one hand, the software approaches optimize performance and energy-efficiency metrics with the implementation of novel algorithms, thus changing the source code of the original implementation. On the other hand the hardware approaches propose optimizations with the design of novel hardware architectures. Our proposed approach is complementary to the above approaches, because it does not propose changes of the algorithms or the underlying hardware architecture, instead it proposes changes in the middleware according to the dynamic inputs of the wireless protocol. Thus, we achieve additional optimizations (by exploiting explicitly the inherent characteristics of the inputs) on top of the ones achieved in the related work.

Finally, the work presented by different groups regarding workload characterization [9], scenario identification [7] and scenario exploitation [8] is very relevant in the context of our proposed approach. The above works focus on defining the characteristics of the run time situations that trigger specific application behavior, which has significant impact on the resource usage of the applications under study. This leads to a specific pattern, which can be exploited as a concrete scenario during the optimization process at the middleware level and reduce significantly the use of the resources in an embedded system.

3 Dynamic Inputs of Wireless Protocols

We have carefully analyzed 18 different network input traces, which come from 18 different 802.11b networks in 5 different buildings of the Dartmouth Campus (over a 17-week period) [9]. These input traces are the result of IPv4 data fragmentation. To make IPv4 more tolerant of different networks the concept of fragmentation was added so that a device could break up the data into smaller pieces. This is necessary when the Maximum Transmission Unit of the Data Link layer is smaller than the datagram size of the Transport layer. For example, the maximum size of a datagram in TCP is 65,535 bytes and the maximum packet size in IEEE 802.11b is 1,500 bytes. In order to send the datagram from TCP, through IPv4, to IEEE 802.11b, the datagram has to be split to smaller packets. Therefore, each 65,535-byte datagram will be split in 1,460-byte packets (assuming that the IP header consumes 40 bytes) and a single datagram will be sent over 45 packets (i.e., 65535/1460 = 44.88). It is very common to have datagram sizes which are not divided exactly by the MTU size and thus the Data Link layer packets have in fact various sizes after the data fragmentation. For each network input trace we have evaluated the occurrence of packet sizes during the transmission and receival of 1,000,000 packets. It is very important to stress that without such an extensive analysis of the aforementioned dynamic inputs, the researcher is most likely to reach wrong conclusions related to the links between the data fragmented inputs and the resulting memory fragmentation (as we will demonstrate in the following subsections). Therefore, we consider the extensive analysis and dynamic input characterization as the most essential part of this publication.

The input pattern actually is a bimodal distribution of run-time memory requests between the Acknowledgement (ACK) and the Maximum Transmission Unit (MTU) packet size. The ACK packets are mostly used in the Transmission Control Protocol (TCP) to acknowledge the receipt of a packet (in order to increase TCP's reliability). Apparently, ACK packets are sent quite often and represent on average 43% of the total packets. Thus, the ACK packet size (i.e., the IP header size of 40 bytes) is the most popular packet size. The MTU is a Data Link layer restriction on the maximum number of bytes of data in a single transmission (i.e., of one 1500 byte packet in the case of IEEE 802.11b). The MTU packet size is the second most popular packet size and averages 26% of the total packets.

It is interesting to note that regardless of the number of packets that we analyzed (i.e., samples from 1000 to 100000 packets as seen in Fig. 1), the percentage of ACK or MTU packets for the average of all the input traces remained almost the same (i.e., less than 2% variance). This was something that we did not anticipate and we interpreted that for a

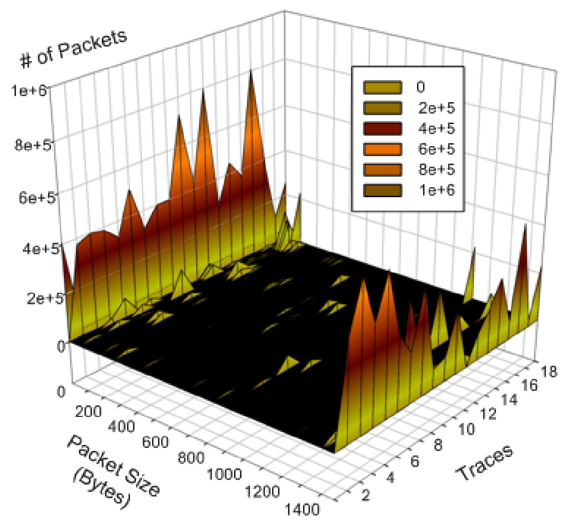

Figure 1. Histograms of all the packet sizes in 18 different network input traces for a sample of 1,000,000 packets

well selected number of traces, even the analysis of a small dynamic input trace can lead to valid conclusions (i.e., just 1000 packets per trace). Also, we had expected to have more MTU sized packets than any other size (but it is actually the ACK packet size that is more frequently requested). As can be seen in Fig. 1, no other sizes are present that represent a significant percentage of the total packets after the data fragmentation. On average, each packet size has 0.02% of the total packets. In all the input traces, more than 1200 different sizes were counted.

Finally, we reach to the conclusion that the pattern present in the dynamic inputs of the wireless networks, only the ACK packet size and the MTU packet size have a significant percentage. We insisted on showcasing this impact that the analysis of a few traces or small samples can have on our evaluation, because in the research domain of Dynamic Memory allocation, the **researchers up to now typically draw their conclusions based on just one or two inputs** and very limited samples (sometimes even synthetic ones [10, 16]).

4 High-performance DM allocation

The datagrams that come from TCP are fragmented into smaller packets and are placed in a queue before they are sent through the Data Link layer to their destination address. The size of this queue and the order that the blocks are placed and forwarded varies from system to system. In terms of packets, the default maximum number of packets that a queue can accommodate in Linux kernel v.2.4 is 100 and in Linux kernel v.2.6 is 1000 [12]. The packets that reside in the queue need to be stored in the memory of the system. It is the responsibility of the DM allocator to give a valid memory address to each request of memory allocation every time that a packet arrives in the queue. It is also the responsibility of the DM allocator to return the memory back to the system, when the packet is forwarded and it does not need any longer to occupy any memory space. The allocation and de-allocation of memory blocks, in order to accommodate the packet needs, eventually leads to (internal and external) memory fragmentation [16]. Various techniques and strategies are used for the design of DM allocators in order to prevent and eliminate internal and external fragmentation. It is important to highlight the software module/algorithms that have the most impact on the design of high-performance DM allocation. For an extensive overview of DM allocator design choices the reader can refer to [3].

Many choices exist in the DM allocator design space and thus the final design can be very complex. The design choices of the state-of-the-art DM allocators are made arbitrarily in respect to dynamic input and the data fragmentation present at the queue of the Network layer and the respective memory fragmentation. In fact, the state-of-the-art in DM allocator designs are usually build according to the memory fragmentation outlook of a wide range of applications, which are completely unrelated to the dynamic inputs of the wireless protocols (the typical set of memory fragmentation benchmarks can be seen in [10]). In this subsection, we are going to propose a novel DM allocator design, which is tuned to the specific memory fragmentation outlook of the inputs of the wireless protocol. Our target is not to replace the standard DM allocator designs, which might be already available in an embedded system. In contrast, our target is to supplement them and thus propagate memory allocation and de-allocation requests, which come from the wireless protocol, to our proposed DM allocator instead of the standard DM allocator. The rest of the memory requests can still be satisfied by the standard DM allocator.

The first step is to translate the data fragmentation present at the Network layer into a pattern of dynamic inputs, which actually sets the memory fragmentation outlook. This is done by evaluating statistically the frequency of any given packet size that has to be placed in the queue and allocated in the memory before it is forwarded. In Section 3, we have shown that the MTU packets and the ACK packets are the most important and frequently used packets sizes for the IEEE 802.11 wireless networks. It is only natural that in the case of other networks the different packet sizes and the MTU will vary. Therefore, the same procedure must be followed in order to evaluate the packet sizes that are used most frequently in each different implementation of the Data Link layer.

The second step is to implement the correct design decisions targeting performance (without compromising memory fragmentation). The statistic results which characterize the dynamic inputs, which are collected at the previous step are used in order to predefine the block sizes that will be used by our proposed DM allocator. **Therefore our proposed optimized design will be centered on the dynamic input pattern (namely the bimodal distribution of sizes).** We propose the following systematic procedure to select and define our DM allocator internals. Note that only the parameters regarding the exact sizes is specific to 802.11b. All the rest of the proposed design choices are generic.

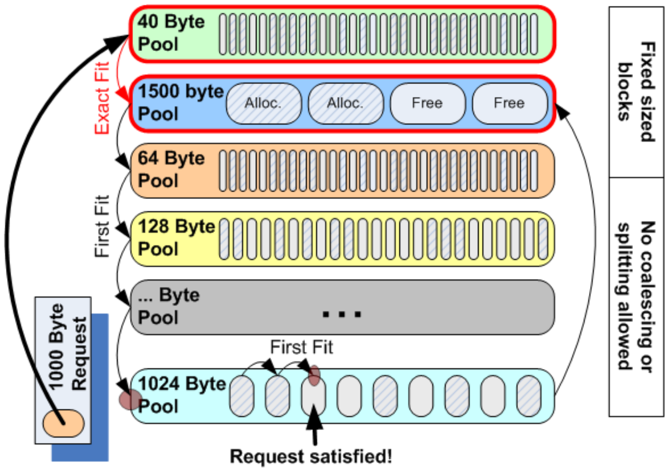

Figure 2. Example: Memory allocation for a 1000 Byte request by our proposed high-performance DM allocator

A.-Proposed blocks: More specifically, we propose to predefine 'special' memory block sizes equal to each packet size that represents at least 10% of the overall packet sizes. The rest of the predefined memory blocks should be power-of-two sizes up to the MTU size. In the case of the IEEE 802.11b, this means that one 'special' predefined memory block of 40 Bytes and one 'special' predefined memory block of 1500 Bytes are present. The rest of the predefined memory blocks should be the 64 Byte block, the 128 Byte block, the 256 Byte block, the 512 Byte block and the 1024 Byte block. Therefore, we are able to satisfy the most popular memory requests without any internal fragmentation and satisfy the remaining less popular requests with reasonable internal fragmentation. Most importantly, the design decision of having fixed sized blocks, predefined at compile-time, gives the performance advantage of not having to calculate the block size for each request at run-time.

B.-Proposed coalescing and splitting blocks: Also, we propose not to use any further splitting or coalescing at the physical level of the memory blocks. We do not need to coalesce any blocks in order to deal with external fragmentation, because we already know the maximum requested block size (i.e., the MTU packet memory request). Additionally, we do not need to split any blocks in order to deal with internal fragmentation, because we have predefined block sizes that prevent most of the internal fragmentation (i.e., the internal fragmentation produced by the popular requests). Most importantly, both the splitting and the coalescing mechanisms are very computationally intensive for the platform hardware, thus slowing down substantially the allocation and de-allocation respectively. Also, these mechanisms have to access the physical memory significantly in order to transform the old block sizes to new ones. The accesses to the memory reduce performance because of the latency associated with each memory access. Especially these choices allow us to save a lot in terms of execution time (as shown in Section 5).

C.-Proposed pools: We propose to use a number of pools equal to the number of the predefined block sizes. In the case of the IEEE 802.11b, we use 2 'special' pools, which hold the 40 Byte and the 1500 Byte blocks, and 5 pools, which hold the 64 Byte, 128 Byte, 256 Byte, 512 Byte and 1024 Byte blocks. This lean memory pool organization is preferred instead of more complex ones, because it allows the fastest access to the memory pool, which will service each request. In the worst case, 8 memory accesses will be needed in order to find the pool which holds the block size relevant to the request. Finally, performance-costly movement of blocks between pools is not used, because it is already made irrelevant with the decision not to support any coalescing or splitting mechanisms.

D.-Proposed fit algorithms: We propose to use among the pools the Exact Fit and First Fit algorithms. The Exact Fit is used only among the 'special' pools, while the First Fit is used among the rest of the pools. Among the blocks, within the pools, we propose the use of only the First Fit algorithm. We have chosen this configuration for our proposed DM allocator, because it requires the least accesses in order to find the memory block to service the memory request. The latency of each memory access is made very explicit in the DM allocator, whose primary concern is to manage the memory, because their number increases exponentially if the wrong combination of pool sizes and fit algorithm is used.

We will illustrate our proposed DM allocator design with an example of a 1000 Byte request (as shown in Fig. 2). Initially, our DM allocator checks if the first 2 'special' pools have blocks with exactly the same size as the request (i.e., Exact Fit). Because our DM allocator did not find a match, it continues and traverses every pool until it finds a pools with blocks big enough to satisfy the request. Eventually, it reaches the 1024 Byte block pool. Inside the pool it traverses with First Fit the first two 1024 Byte blocks, which are allocated and returns the third block, which is free and satisfies the request. Note that a request for a 1200 Byte block would check finally the 1500 Byte block pool and it would be satisfied with a 1500 Byte block.

5 Performance Evaluation and Comparisons

In this section, we will evaluate our proposed DM allocator and compare it with the state-of-the-art Linux DM allocator (which is considered the best DM allocator [10]). We will provide the simulated results of both of them for the input traces that were analyzed in Section 3. We show the results for the Linux IPv4 data fragmentation of TCP datagrams, which are to be sent with the Linux implementation of the IEEE 802.11b wireless LAN protocol. The TCP inputs for the IPv4 data fragmentation are taken from the 18 different input network traces of the Dartmouth Campus [9]. All our simulations are made for a queue size of 1000 elements, which is the default value for the Linux kernel 2.6. All the results regard the allocations needed to fill and empty the queue, thus they correspond to 1000 memory allocations and 1000 memory de-allocations. We have developed a framework and automated tool support [13] to customize and specialize easily any DM allocator and thus validate the effectiveness of our proposed methodology. We provide results for the performance (i.e., execution time), memory footprint, memory accesses and energy consumption of our proposed DM allocator and the Linux DM allocator. The energy estimations are made using the model in [14]. For the energy measurements we assume a flat memory hierarchy with a single on-chip SRAM memory.

In Fig. 3, we evaluate the performance of our proposed DM allocator. On the one hand, the execution-time needed by our proposed DM allocator rests always just below 100 milliseconds with very small variations. On the other hand, the Linux DM allocator needs on average more than 200 milliseconds and its performance varies considerably from input trace to input trace. On average, our DM allocator has 60.18% better performance than the Linux DM allocator. The splitting mechanism is used most of the time in the Linux DM allocator, because for 40 Byte requests it can only split predefined 64 Byte blocks and for 1500 Byte requests it can only split predefined 4096 Byte blocks. The situation is made even worse (in terms of performance) in Linux during the de-allocation of blocks, because all those split 64 Byte and 4096 Byte blocks will have to be coalesced again. Also, our choice to employ just 7 pools (instead of the 128 pools present in the Linux DM allocator) render the traversing of the pools much faster. Finally, the choice of the First Fit algorithm reduces the accumulated latency of all the memory accesses that are needed to traverse the pools and the blocks inside them. This is much faster than the Best Fit algorithm in the Linux DM allocator, which needs to search all the pools and all the free memory blocks inside a pool before it satisfies a request.

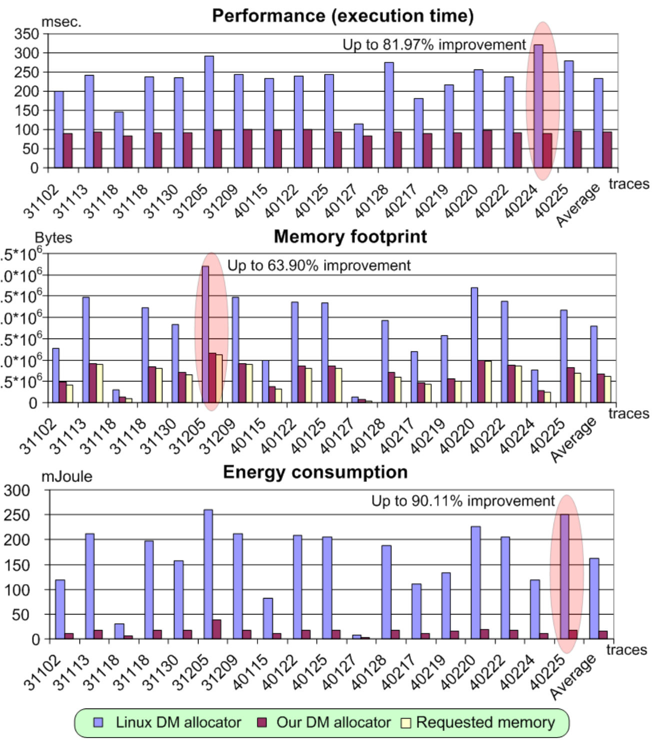

Figure 3. DM allocator performance, memory footprint and energy consumption

In Fig. 3, we also evaluate the memory footprint of our proposed DM allocator. On the one hand the memory footprint needed by our proposed DM allocator is on average 671,210 Bytes. This is not much more than the average requested memory, which is 621,172 Bytes. Therefore, our prosed DM allocator has 8.05% memory fragmentation. On the other hand, the Linux DM allocator proves very inefficient with an average memory footprint of 1,793,611 Bytes, thus demonstrating an impressive 188.74% of memory fragmentation. On average, our DM allocator has 62.58% less memory footprint than the Linux DM allocator. We also evaluate the energy consumption of our proposed DM allocator. On the one hand, the energy consumed by our proposed DM allocator is on average 16,072 millijoule. On the other hand, the Linux DM allocator needs on average 162,488 millijoule. On average, our DM allocator consumes 90.11% less energy than the Linux DM allocator due to two factors. The first factor is the implementation of design choices that reduce memory accesses, which themselves consume energy. The second factor of the reduction is the use of smaller memories by our proposed DM allocator. Our DM allocator is able to use smaller memories than the Linux DM allocator because its memory footprint is smaller. Consequently, smaller memories are more energy efficient and consume less energy per access. The combination of these two factors gives the impressive reduction of one order of magnitude.

However, a trade-off exists, namely the object code of the DM allocator itself. Because we propose not to replace the standard DM allocator the size of the object code of our DM allocator must be added on top of the object code of the standard DM allocator. Nevertheless, the 11.4 KByte object code increase that we have, is relatively small considering the reduction of memory footprint that our DM allocator provides, thus it is a very desirable trade-off.

6 Conclusions

The dynamic input at the Data Link Layer is the result of run-time data fragmentation at the Network layer. The pattern of this dynamic input results to a pattern of memory fragmentation. If the Dynamic Memory allocator (at the middleware) is not fine tuned to this pattern of the dynamic input, it will have significant internal and external fragmentation and it will have reduced performance. In this paper, we focus on the design of high-performance Dynamic Memory allocators, which are sensitive on the dynamic input pattern and base their design on the statistical distribution of the memory blocks that are requested by the data fragmentation mechanism. We show that in the case of IPv4 data fragmentation of TCP datagrams to IEEE 802.11b packets, the ACK size and the MTU size dominate the design and play a significant role in the fine tuning of the proposed DM allocator. Finally, we show significant improvements over the state-of-the-art Linux DM allocator with a small object code size increase trade-off.

7 Acknowledgements

This work is partially supported by the E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031. We want to thank David Atienza (DACYA UCM, Madrid, Spain) and Christophe Poucet (IMEC, DESICS group, Leuven, Belgium) for their help and support in the development of the DM allocator library and profiling framework.

References

[1] Ipv4 documentation - rfc 791. `http://www.ietf.org/rfc/rfc791.txt`.

[2] Rfc 2675 ipv6 and jumbograms. `http://rfc.sunsite.dk/rfc/rfc2675.html`.

[3] D. Atienza, J. M. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. volume 11, pages 465–489, New York, NY, USA, 2006. ACM.

[4] G. Attardi, T. Flagella, and P. Iglio. A customizable memory management framework for c++. *Software Practice and Experience*, 28(11):1143–1183, 1998.

[5] E. Berger, B. Zorn, and K. McKinley. Composing high-performance memory allocators. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 114 – 124, Snowbird, Utah, 2001.

[6] P. Bhagwat, C. Perkins, and S. Tripathi. Network layer mobility: An architecture and survey. In *IEEE PERSONAL COMMUNICATIONS*, volume 3, pages 54 – 64, 1996.

[7] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2 2003.

[8] S. V. Gheorghita, T. Basten, and H. Corporaal. Intra-task scenario-aware voltage scheduling. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, New York, NY, USA, 2005. ACM Press.

[9] T. Henderson, D. Kotz, and I. Abyzov. The changing usage of a mature campus-wide wireless network. In *Proceedings of the Tenth Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 187–201. ACM Press, September 2004.

[10] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the International Symposium on Memory Management*, Vancouver, British Columbia, October 1998.

[11] C. E. Jones, K. M. Sivalingam, and P. A. J. C. Chen. A survey of energy efficient network protocols for wireless networks. In *WIRELESS NETWORKS*, 2001.

[12] F. Li, M. Li, R. Lu, H. Wu, M. Claypool, and R. Kinicki. Tools and techniques for measurement of ieee 802.11 wireless networks. In *The Second International Workshop On Wireless Network Measurement (WiNMee)*, 2006.

[13] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, D. Soudris, and J. M. Mendias. Automated exploration of pareto-optimal configurations in parameterized dynamic memory allocation for embedded systems. In *Proceedings of DATE '06: Proceedings of the conference on Design, automation and test in Europe*, page 874, Germany, 2006.

[14] A. Papanikolaou, M. Miranda, F. Catthoor, H. Corporaal, H. D. Man, D. D. Roest, M. Stucchi, and K. Maex. Global interconnect trade-off for technology over memory modules to application level: case study. In *SLIP '03: Proceedings of the 2003 international workshop on System-level interconnect prediction, ACM press*, 2003.

[15] P. G. Paulin, F. Karim, and P. Bromley. Network processors: A perspective on market requirements, processor architectures and embedded s/w tools. In *Proceedings of Design, Automation and Test in Europe*. IEEE Press, 2001.

[16] P. R. Wilson, M. S. Johnstone, and et al. Dynamic storage allocation, a survey and critical review. In *International Workshop on Memory Management*, Kincross, Scotland, UK, 1995. Springer Verlang LNCS.