ADVANCED TECHNOLOGIES AND APPLICATIONS SESSION
INVITED PAPER

SCRATCHPAD MEMORY UNIT IN HYBRID CONTROL-FLOW AND DATAFLOW ARCHITECTURES

Nenad Korolija[1]*,
Svetlana Štrbac-Savić[2],
Borislav Đorđević[2]

[1]School of Electrical Engineering,
University of Belgrade,
Belgrade, Serbia

[2]Academy of Technical and Art Applied
Studies Belgrade,
Belgrade, Serbia

Abstract:

Dataflow architectures offer superior performance compared to control-flow architectures under certain conditions. This paper focuses on memory organization of a hybrid control-flow and dataflow architecture which guaranties that memory allocation can be accomplished in a predictable time. Mapping logical addresses into physical ones and accessing local memory in constant time is achieved using special address translation hardware. The memory organization is based on Buddy-system. It allows allocating arbitrary amounts of memory and prefetching data while it is being accessed by control-flow and dataflow hardware.

Keywords:

Dataflow architectures, Control-flow architectures, Scratchpad memory, Buddy-system.

INTRODUCTION

Real-time systems are a form of operating systems with a special purpose. They are employed when there are strict time constraints for job execution. A real-time system is considered to function correctly if and only if it returns a correct result respecting precisely defined time constraints.

In order to reliably determine the longest program execution time, many real-time systems are designed in such a manner, so that the memory access time is calculated as the time required to retrieve data from the main memory, assuming that the data will never be found in the cache memory. Some of the solutions involve the use of a cache entry locking mechanism. This ensures that the data, once loaded and locked in one cache entry to be written by one thread, cannot be changed by another thread. Another way to predict the maximal execution time is based on memory partitioning. In this case, statically sized partitions at boot time, and/or dynamic partitions can be used.

Correspondence:
Nenad Korolija

e-mail:
nenadko@etf.rs

DOI: 10.15308/Sinteza-2023-159-165

Under certain circumstances, dataflow architectures are capable of executing a higher number of instructions per second compared to control-flow architectures [1, 2]. However, there are a lot of challenges. Programming dataflow architectures is considered to be reasonably harder compared to programming control-flow architectures [3]. Despite that, there are a lot of algorithms implemented for dataflow architectures [4], as well as methods to automate the translation of control-flow software into the dataflow [5, 6]. Placing both control-flow and dataflow components with shared memory at the same chip die requires special techniques [7-10], while it reduces the lifespan of a chip to the minimum lifespan of its components [11]. The applicability of the architecture is still being investigated [12]. Scheduling the execution of dataflow and control-flow jobs introduces more constraints compared to scheduling for control-flow architectures [13]. The granularity of jobs for which is justified to utilize dataflow hardware is still a research topic [14]. In short, there is a long way from an algorithm to the execution on a special chip [15].

This paper examines the allocation and the deallocation of memory, as well as local memory access mechanisms in hard real-time systems for the purpose of hybrid control-flow and dataflow processors.

2. PROBLEM DEFINITION

The aim of the work is to solve the problem of local memory fragmentation and memory access speed, so that this kind of memory model can be used in hybrid architectures. It is necessary to foresee:

- appropriate structures that store data on free and reserved memory blocks, such that an appropriate amount of free memory can be found in a predictable and relatively short time interval
- appropriate mechanisms that can access a local memory in a predictable time.

In addition to this, it is necessary to provide methods by which it is possible to load data into local memory before starting a program execution, if this is possible. Otherwise, it should be loaded during the execution.

3. EXISTING SOLUTIONS AND THEIR ANALYSIS

In recent decades, a lot of work has been done on solving the problem of cache memory partitioning in the case of multiprocessor systems. Most of the solutions represent either techniques for sharing resources by logically dividing cache memories [16-22] or techniques for assigning cache memory partitions in the case where a system supports private cache memories [23-26]. Unfortunately, various techniques introduce problems of unfairness [27], so-called trashing [28], and quality of service problems [29].

One of the solutions to the aforementioned problems is the allocation of cache partitions of requested sizes only in certain time intervals [30]. The justification for this approach lies in the following fact. If allocating a cache partition smaller than the partition size significantly reduces the efficiency of the system, the standard partition is not suitable. In order to avoid an unfair solution, in which some threads have the required amount of cache memory, while others have significantly less, the cache memory can be divided so that each thread has the required cache partition size only at certain intervals.

Another solution was proposed in the paper [29]. The main idea is to partition the set-associative cache, so that threads access always the same sets.

In each of the previously described cases, there is still the problem of fragmentation or the limitation of the amount of local memory that can be allocated to one thread. In the case of using fixed-length partitions, the problem of internal fragmentation occurs. If one thread was allowed to own exactly one partition, this would unnecessarily introduce a limit on the amount of memory that a thread can possess. If each thread was allowed to have a variable number of partitions, keeping an appropriate structure that would store data about occupied partitions for each thread would require additional memory, as well as the time needed to access this data.

4. ASSUMPTIONS

Before solving the problem of local memory organization, appropriate assumptions are listed:

- it is possible to allocate any amount of free memory, which implies relatively little internal fragmentation

- it is possible to utilize the portion of a memory reserved for one thread, while freeing the space necessary for another thread
- it is possible to wait for the reservation of memory requested by one thread until the execution of another thread is finished, as long as the waiting time is predictable so that the use of memory blocks can be scheduled.

5. PROPOSED SOLUTION

This paper proposes a solution to the problem based on the so-called Buddy System [31] mechanism for storing data in memory. Figure 1 depicts an example logical organization of a reserved memory consisting of multiple blocks, each of them with the size 2^n bytes, where n is a positive integer number. It goes without saying that each request for a certain amount of memory can be represented as a sum of requests for reserving memory blocks of size 2^n, where each block is of a different size.

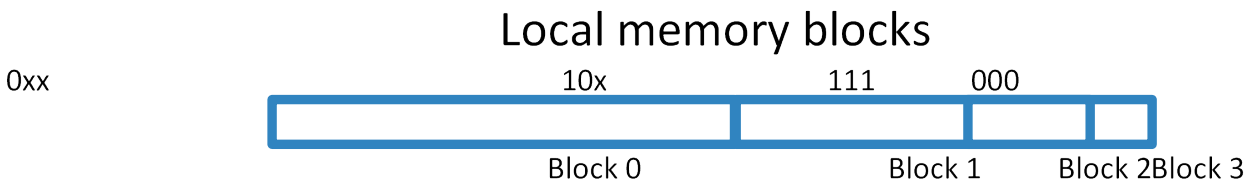

Figure 1 - Modeling a reserved memory as a sum of blocks.

Reserving memory for the requested sequence of blocks starts by checking whether the requested amount of memory is available in the system. Reserved memory consists of a sum of reserved blocks. We can also call them requested blocks. Then, for each of the blocks, the place in the local memory has to be found or freed. The following three cases can occur:

- there is a free block of the requested size
- there is a free larger block
- there is no block of sufficient size, but the total amount of free memory is larger than the requested block size.

In the case there is a free block of the requested size, one of such blocks must be declared as occupied, and the address of the beginning of the newly occupied block is written into the resulting vector of the memory reservation function. Otherwise, if a larger free block exists than the requested one, that block is divided into smaller ones, one of which is declared occupied.

In the case that there is no large enough block available in the system, but there is enough free space in the system, another function is called to move blocks in the local memory in such a way that the maximum amount of data that may be required to move in order to free up space for this block is predictable. First, a block of the required size is found, for which the cost of moving the occupied portion of data elsewhere is estimated to be minimal.

It should be noted that this does not mean that the highest percentage of the block is empty. For example, moving five blocks of the smallest size may require at most five moves of the amount of data of one of the smallest blocks. On the other hand, moving a block of size equal to four of the smallest blocks may involve moving a block of size equal to two smallest blocks, as well as moving another block of the smallest size, to make room for a block of size equal to four smallest blocks. Then it is necessary to move the block for which the memory is reserved, i.e. another moving of the size equal to four smallest blocks is needed. Moving a block of the size equal to two smallest blocks, it is again necessary to make room for it, which may involve moving one block of the smallest size, and then moving a block for which the memory is freed, ie. moving the block of size equal to two smallest blocks. In the described way, it would be necessary to move eight blocks of the smallest size, which is significantly more than moving five smallest blocks. Therefore, the cost of moving five blocks of the smallest size could be assigned the number five, while the cost of moving a block of size equal to four blocks of the smallest size could be assigned the number eight.

After finding the block that is found to have the lowest moving cost, it is necessary to recursively call the same function to free space in the local memory for each of blocks contained in it by move contained blocks.

The function for freeing the local memory processes the vector passed to the function as an input parameter, and is responsible for freeing a memory for each of the blocks corresponding to the components of this vector. If, by freeing a memory occupied by any of the blocks, it is determined that the adjacent block of the same size is empty with which it can form a bigger block, it is necessary to declare these two blocks as one single free block of twice the size. Again, the resulting bigger free block has an empty neighborhood block of the same size, with which it can form a bigger block, these two blocks must be merged again in order to become one free block. This procedure is repeated until the previously described condition is not met.

In order to support access to a memory with such organization, it was necessary to provide the mechanism for mapping generated logical addresses into the physical ones. Each thread accesses its local memory as if it was a contiguous memory of certain size. Following actions need to be performed in order to enable mapping generated addresses into physical ones:

- determine which block the data belongs to
- determine the displacement relative to the beginning of the block.

Block *i*, where *i* is a positive integer, is exactly twice the size of block *i*+1. If the heaviest bit of the generated address is set to 1, block 0 is accessed. In this case, the remaining bits of the generated address are taken as an offset relative to the beginning of that block.

In the case the bit of the highest weight of the generated address is set to 0, the next bit is observed. If it is set to 1, block 1 is accessed. In that case, the remaining bits of the generated address are used to determine the displacement. This way, it is determined which block needs to be accessed, as well as the offset relative to the beginning of that block. In general, the first bit of the generated address that is found to be set (equal to 1) determines the block address, and the bits to the right of this bit represent the offset.

This way, it is possible to transform the generated address into the physical address without using an adder, that is, only by choosing which bits should be taken from the block address, and which bits should be taken from the generated address. The process of transforming the generated into physical addresses is depicted in Figure 2. The block 0 size is equivalent to one half of the memory size, and, as such, can be placed either in the first half, or in the second half of the memory. Therefore, the starting address of the block has only 1 bit that is important, while other bits are equal to 0, and can therefore be ignored (don't need to be stored). Similarly, block 1 address contains only the first 2 important bits, while the rest of the bits are considered to be equal to 0. In general, block i address has exactly the first *i*+1 important bits. In order to determine the physical address, these important bits are taken from the block starting address, while the remaining bits are taken from the generated address, as they represent the offset from the beginning of the block.

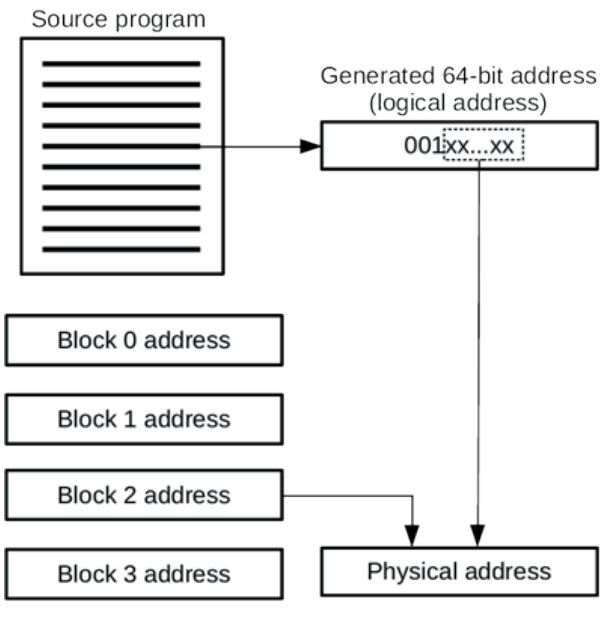

Figure 2 - The process of transforming logical into physical addresses.

Once the memory has been reserved for a thread, it is possible to load the data required by a thread into the local memory before the execution of the thread. It is also possible to load the data into one of the obtained memory blocks, while the remaining blocks have not been freed.

6. MEMORY RESERVATION ANALYSIS

In this chapter, an analysis of the proposed solution for reserving a memory is given, with the aim of enabling the proper configuration of a system, so that the internal fragmentation is as small as possible, as well as the time required to provide the required amount of data. First, the theorem and the corresponding proof are presented.

Theorem 1: In order to provide the space for a block whose size is equal to 2^n smallest data blocks, it is necessary to move, in the worst-case, $n*2^{n-1}$ smallest data blocks.

Proof: First, we can assume that the total amount of free memory is always greater than the requested amount of memory. Otherwise, reserving the requested amount of memory is not possible.

Let us assume that the value of the number n is equal to 1, and that the smallest block size is equal to 1 byte.

If we want to provide the space for 2^1 bytes, it is in the worst case necessary to move 1 byte from one place in the memory to another. If we set the value of the number *n* to 1 in the formula given in the theorem, we get the same result.

By applying mathematical induction, if we assume that the formula given in the theorem is correct for some arbitrary value of the number *n*, we can derive how many of the smallest data blocks need to be moved in the event that the required amount of memory is 2^{n+1} bytes.

It is known that it is necessary to move, in the worst case, $n*2^{n-1}$ smallest data blocks, in order to make room for 2^n data blocks.

In addition to this, in the worst case, it is necessary to move another block of the size equal to the size of 2^n smallest data blocks, in order to provide space for a data block of size equal to the size of 2^{n+1} smallest data blocks.

In order to provide space for 2^n data blocks, it is necessary to move, in the worst case, $n*2^{n-1}$ smallest data blocks.

Summing up the necessary moves of the smallest data blocks, we obtain the result of Equation 1.

$$2^n+n*2^{n-1}+n*2^{n-1}=2^n+n*2^n=(n+1)*2^n \quad (1)$$

Equation 1 - Number of blocks needed to be moved in the worst case.

As we have shown that the theorem is true in the case the value of the number *n* is set to 1, and then, starting from the assumption that it is true for some value of the number *n*, that it is also true for the value n+1, it has been proven that the theorem is true.

Therefore, the size of the smallest block should be chosen, so that the system has the best performance. The optimal value of the size of the smallest block can be determined empirically, having in mind the internal fragmentation and the control logic needed for handling addresses of blocks.

7. RESULTS

By analyzing the proposed solution, it was determined that, in order to provide contiguous space for the 2^n smallest data blocks, in the worst case $n*2^{n-1}$ smallest data blocks must be moved. The simulator implemented in the programming language C++ enables monitoring of block movement when freeing local memory. The simulation results confirm that the number of necessary moves of smallest blocks cannot exceed the value given in a Theorem 1. As an example, a system was implemented for which the ratio of the number of bytes that need to be moved and the required amount of local memory in the worst case is four, which corresponds to the value of the number n set to eight in the formula. It should be noted that this factor is proportional to the number *n*.

8. CONCLUSION

By analyzing amounts of memory and hardware resources needed to store data structures intended for data access and implement data access in hardware, as well as by comparing the data access times, from all the analyzed systems, the Buddy System was chosen due to the deterministic time needed for memory allocation and its simplicity. Appropriate C++ simulator was implemented, proving the validity of theoretical analysis.

Bearing in mind that the time required to access the main memory of a computer is far greater than the time required to access the local memory (e.g. cache or scratchpad), it can be considered that the proposed solution effectively solves the problem of fragmentation and enables sharing the local memory between control-flow and dataflow hardware.

Further research directions relate to determining the optimal relationship between the minimal block size and the amount of memory that can be reserved, so that the access hardware is relatively small, as well as internal fragmentation, while the size of the largest block that can be reserved does not limit the execution of high performance algorithms.

9. REFERENCES

[1] R. Trobec, R. Vasiljević, M. Tomašević, V. Milutinović, R. Beivide, and M. Valero, "Interconnection networks in petascale computer systems: A survey," ACM Computing Surveys (CSUR), vol. 49, no. 3, pp. 1-24, 2016.

[2] V. Milutinović, B. Furht, Z. Obradović, and N. Korolija, "Advances in high performance computing and related issues," Mathematical problems in engineering, 2016.

[3] J. Popovic, D. Bojic, and N. Korolija, "Analysis of task effort estimation accuracy based on use case point size," IET Software, vol. 9, no. 6, pp. 166-173, 2015.

[4] N. Trifunovic, V. Milutinovic, N. Korolija, and G. Gaydadjiev, "An AppGallery for dataflow computing," Journal of Big Data, vol. 3, pp. 1-30, 2016.

[5] V. Milutinovic, J. Salom, D. Veljovic, N. Korolija, D. Markovic, L. Petrovic, ... and L. Petrovic, "Transforming applications from the control flow to the dataflow paradigm," DataFlow Supercomputing Essentials: Research, Development and Education, pp. 107-129, 2017.

[6] V. Milutinović, N. Trifunović, N. Korolija, J. Popović, and D. Bojić, "Accelerating program execution using hybrid control flow and dataflow architectures," In 2017 25th Telecommunication Forum (TELFOR), IEEE, pp. 1-4, November 2017.

[7] V. Milutinović, E. S. Azer, K. Yoshimoto, G. Klimeck, M. Djordjevic, M. Kotlar, ... and I. Ratkovic, "The ultimate dataflow for ultimate supercomputers-on-a-chip," for scientific computing, geo physics, complex mathematics, and information processing, In 2021 10th Mediterranean Conference on Embedded Computing (MECO), IEEE, pp. 1-6, June 2021.

[8] V. Milutinović, M. Kotlar, I. Ratković, N. Korolija, M. Djordjevic, K. Yoshimoto, and M. Valero, "The ultimate data flow for ultimate super computers-on-a-chip," In Handbook of Research on Methodologies and Applications of Supercomputing, IGI Global, pp. 312-318, 2021.

[9] N. Korolija and K. Milfeld, "Towards Hybrid Supercomputing Architectures," Journal of Computer and Forensic Sciences, vol. 1, no. 1, pp. 47-54, 2022.

[10] D. Miladinović, M. Bojović, V. Jelisavčić, and N. Korolija, "Hybrid Manycore Dataflow Processor," 9th IcETRAN Conference 2022, Novi Pazar, Republic of Serbia, June 6-9, 2022.

[11] K. Huang, Y. Liu, N. Korolija, J. M. Carulli, and Y. Makris, "Recycled IC detection based on statistical methods," IEEE transactions on computer-aided design of integrated circuits and systems, vol. 34, no. 6, pp. 947-960, 2015.

[12] J. Popović, V. Jelisavčić, and N. Korolija, Hybrid Supercomputing Architectures for Artificial Intelligence: Analysis of Potentials, AAI 2022 Conference, Kragujevac, Serbia, May 19-20, 2022.

[13] N. Korolija, D. Bojić, A. R. Hurson, and V. Milutinovic, "A runtime job scheduling algorithm for cluster architectures with dataflow accelerators," Advances in computers, vol. 126, pp. 201-245, 2022.

[14] N. Korolija, B. Furht, and V. Milutinović, "Fine Grain Algorithm Parallelization on a Hybrid Control-flow and Dataflow Processor," 2023.

[15] V. Milutinović, M. Kotlar, J. Salom, S. Stojanović, Ž. Šuštran, A. Veljković, ... and R. R. Hurson, "VLSI for SuperComputing: From applications and algorithms till masks and chips," 2022.

[16] A. Šmelko, M. Kruliš, M. Kratochvíl, J. Klepl, J. Mayer, and P. Šimůnek, "Astute Approach to Handling Memory Layouts of Regular Data Structures," In Algorithms and Architectures for Parallel Processing: 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10–12, 2022, Cham: Springer Nature Switzerland, Proceedings pp. 507-528, January 2023.

[17] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," IEEE Micro, vol. 20, no. 2, pp. 71–84, 2000.

[18] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha, "A Scalable Architecture Based on Single-Chip Multiprocessing," Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27), 2000.

[19] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy, "IBM Power4 system microarchitecture," IBM Journal of Research and Development, vol. 46, no. 1, pp. 5–26, 2002.

[20] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled CMPs," Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32), 2005.

[21] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: A complexity analysis," In Proceedings of the SCIzzL, vol. 6, pp. 89-96, September 1996.

[22] A. Ngom, I. Stojmenovic, and V. Milutinovic, "STRIP-a strip-based neural-network growth algorithm for learning multiple-valued functions," IEEE Transactions on Neural Networks, vol. 12, no. 2, pp. 212-227, 2001.

[23] E. Speight, H. Shafi, L. Zhang, and R. Rajamony, "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors," Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32), 2005.

[24] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication and Capacity Allocation in CMPs," Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32), 2005.

[25] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," Proceedings of the 33th Annual International Symposium on Computer Architecture (ISCA-33), 2006.

[26] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39), 2006.

[27] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT-13), 2004.

[28] P. J. Denning, "Thrashing: Its Causes and Prevention," AFIPS 1968 Fall Joint Computer Conference, vol. 33, pp. 915–922, 1968.

[29] A.M. Molnos, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, "Compositional Memory Systems for Multimedia Communicating Tasks", Proceedings of the conference on Design, Automation and Test in Europe, vol. 2, 2005.

[30] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," Proceedings of the 21st annual international conference on Supercomputing, 2007.

[31] P. Purdom and S. Stigler, "Statistical Properties of the Buddy System," Journal of the ACM (JACM), vol. 17, no. 4, pp. 683-697, 1970.