SPATIAL ISOLATION ON REALTIME HYPERVISOR USING CORE-LOCAL MEMORY

Daisuke Yamaguchi, Takumi Yajima, Chen-Yi Lee, Hiromasa Shimada, Yuki Kinebuchi and Tatsuo Nakajima
Distributed and Ubiquitous Computing Laboratory, Waseda University, Tokyo, Japan

Keywords: Operating Systems, Embedded Systems, Security, Hypervisor, Virtualization, Multicore Environment, Core-local Memory.

Abstract: Recently, the software of embedded systems grows increasingly complicated due to controversial needs of both rich functionalities and strict interrupt responsiveness. In order to deal with it, realtime virtualization technology for embedded systems is attracting interests. Virtualization allows multiple operating systems to run concurrently with minimal modifications, thus reduce the engineering cost. However, as the security of embedded systems getting more concerns in these days, current design of realtime hypervisor often makes it difficult to ensure the security without hardware virtualization support which is not widely available in the world of embedded systems. In this paper, we introduce Secure Pager which utilizes a common hardware design called core-local memory combined with check-sum based protections to enforce the spatial isolation without specific hardware virtualization support.

1 INTRODUCTION

In recent years, virtualization technology has been proved as an effective approach to improve the overall system performance and eliminate the management cost. Besides, it is also an attractive approach for the embedded system development. In order to effectively virtualize the hardware resource, most of previous works either rely on hardware virtualization support, such as Intel VT and AMD-V, or require a large amount of modification to guest OSes. However, since only a limited number of embedded architectures currently support virtualization, developers have to port the existing software to processors with such features, which incurs non-trivial engineering cost as a result. One of our goals is to provide the same functionality while keeping the amount of modification to guest OSes as small as possible. On the other hand, high responsiveness is another aspect that should be maintained. As a result, current hypervisors with realtime responsiveness (so called *realtime hypervisor*) often do not support MMU virtualization and have only one privileged access level among all guest OSes. Nevertheless, there is growing concerns in security issues on embedded system nowadays. For example, there is an increasing number of both security holes in the Android system and the malware targeted on it, which made it difficult for realtime hypervisors to ensure the security as the compromised guest OS being able to attack the memory image of other guest OSes.

In this paper, we propose the solution utilizing core-local memory for spatial isolation. Core-local memory is an SRAM integrated in the processor core which was originally designed for high-speed data manipulation, such as multimedia encoding/decoding. Moreover, multicore and many-core processors in recent years are expected to utilized the core-local memory in order to avoid bus contention. Therefore we could conclude that core-local memory is more common than hardware virtualization support. Several current architectures having core-local memory are listed in Table 1.

There are four challenges in our approach:

- Require minimal modification to the guest OS.
- Utilize commonly supported hardware rather than any other special hardware support.
- Keep the interrupt latency as small as possible thus to ensure the responsiveness.
- Keep the implementation simple and possible to be applicable to other platforms.

Yamaguchi D., Yajima T., Lee C., Shimada H., Kinebuchi Y. and Nakajima T..
SPATIAL ISOLATION ON REALTIME HYPERVISOR USING CORE-LOCAL MEMORY.
DOI: 10.5220/0003906704150421
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 415-421
ISBN: 978-989-8565-00-6
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

Table 1: Architectures with core-local memory.

Architecture	Core-local memory
ARM11	4 to 256KB
MIPS32 4KE Family	Up to 1MB
SuperH	Instruction: 8KB Data: 16KB

2 BACKGROUND

2.1 SPUMONE: The Realtime Hypervisor

SPUMONE (Kanda et al., 2008) is a lightweight realtime virtualization layer for embedded systems. The original goal of SPUMONE is to realize both the realtime constraints and rich functionality through combining RTOS (realtime OS) and GPOS (general-purpose OS) running on the same system concurrently. In order to achieve this goal, SPUMONE only virtualizes the processor state and has only one privileged access levels. SPUMONE also keeps the amount of modification to guest OSes as small as possible. In addition, SPUMONE has a simple design. The source code of SPUMONE is around 5K LOC, which is far smaller than most of the modern general-purpose hypervisors, such as *Xen* (Barham et al., 2003), secure-oriented *BitVisor* (Shinagawa et al., 2009) and *SecVisor* (Seshadri et al., 2007).

On the other hand, the simplicity of design creates some potential security holes. For example, a fundamental problem is that there is no spatial isolation among guest OSes and SPUMONE, which means that all images are allocated in the shared main memory without any protection. While the guest OSes and SPUMONE are running at the same privilege level as mentioned above, once a guest OS get compromised, it is able to attack the other guest OS and SPUMONE directly through memory access.

In this paper, we enhance the security of such type of realtime hypervisor through *Secure Pager*, which ensures the spatial isolation of shared main memory. The mechanism of Secure Pager is described in section 4.

2.2 Related Work

Works on the memory protection in hypervisors has been extensively carried out, often with specific hardware support. *SecVisor* is a tiny hypervisor which ensures Linux's kernel image integrity against code injection attacks such as kernel rootkits. It protects the kernel image by virtualizing the physical memory and managing page tables. However, those protections are established by CPU-based virtualization (*AMD's Secure Virtual Machine technology* (Advanced Micro Devices, 2011)), which is hardly equipped in the embedded systems. In addition, this mechanism is capable of protecting one single guest OS only which means it is difficult for SecVisor to meet the needs of recent advanced embedded systems.

SafeG (Sangorrin et al., 2010) is a dual-OS monitor which enables RTOS and GPOS to run concurrently on the same machine. Because SafeG aims on embedded systems, the realtime constraints are guaranteed. It also protects RTOS memory and devices from illegal attack of GPOS. However, the SafeG architecture takes advantage of CPU-based virtualization (*ARM TrustZone* (Alves and Felton, 2004)), which could not be applicable to old hardware architectures. Furthermore, it hasn't been ported to a multicore environment. Unfortunately, it causes the vulnerability of cache attacks. Because caches are not separated between RTOS and GPOS, non-trust GPOS can affect the performance of secured RTOS by flushing the cache.

On the other hand, the utilization of core-local memory is getting more attention these days. There is already a significant amount of study on the benefits of using such kind of on-chip memory. (Banakar et al., 2002) compared caches and on-chip memories from various aspects, with the results revealing that the average area-time reduction was 46% when caches are replaced by the on-chip memories, and the average reduction of energy consumption was 40%.

Single-chip Cloud Computer (Held, 2010) invented by Intel Labs also adopted software-managed on-chip memories. Instead of using hardware-managed caches, it enables application programmers with flexible management of on-chip data. The reduced energy consumption of on-chip memory also meets the the limited power budget on many-core. The developers of *Single-chip Cloud Computer* believes that "software managed coherency on non-coherent many-core is the future trend" (Jim Held, SCC Symposium Materials, 2010, Software-Managed Coherency, §6).

Cell Broadband Engine(*Cell/B.E.*) (Shimizu et al., 2007) is a microprocessor architecture which has strong hardware-based security mechanisms. *Cell/B.E.* is a multicore architecture with 9 cores on each processor, while one of them is a general purpose core called *PowerPC Processor Element*(PPE) which controls the other cores, and the others are called *Synergistic Processor Element*(SPE) which plays the major role in computation. Each SPE core has its own local storage which is physically inaccessible from other cores when they are in isolation mode. Because of this absolute hardware-based isolation, even the operating system can not access the data stored in the locked local storage. In addition, before loading data from main memory to local storage, authentication is required to check whether the data has been modified or not, by means of hardware key and a cryptographic algorithm. However, establishing authentications by hardware makes the system inflexible and hard to update. By replacing the hardware-based authentication mechanisms with minimal software, we believe that more flexible and deployable secure environment can be established in various modern processors of embedded systems.

3 ASSUMPTIONS

First, we assume that multiple guest OSes are running on top of the virtualization layer in a multicore processor environment, with the guest OSes and virtualization layer running at the same privileged level. In this model, the security of guest OSes is not ensured and thus any of them may be compromised by rootkits. Then the compromised guest OS can perform attacks on other guest OSes and even the virtualization layer directly through normal memory access. In this situation, any internal secure module of virtualization layer which resides in main memory becomes failed to protect guest OSes and virtualization layer itself. For simplicity, because we only focus on the runtime protection, we assume that the booting process of whole system would not be compromised.

We created the isolated and secure space in such environment by utilizing the core-local memory, combined with a checksum-based integrity management scheme. It comes at the cost of not supporting MMU virtualization, which indicates that our implementation can protect only an OS which is not utilizing the address translation provided by MMU. (However, many of embedded OSes do not utilize such feature.) We implemented on *TOPPERS* (TOPPERS Project, 2004), which is a simple OS without virtual memory management.

4 DESIGN

4.1 Core-local Memory based Protection

As mentioned in section 1, we utilize the hardware called core-local memory in a multicore processor to ensure the memory space isolation. The idea is first proposed by (Kinebuchi et al., 2010) to ensure the security of hypervisors on general-purpose computers rather than embedded systems. Since the core-local memory is inaccessible from the other cores, we leverage on this characteristic to protect target guest OS. Fig 1 shows the behavior of core-local memory. In this situation, two guest OSes are running concurrently. Guest OS #1 is running on core 0 while guest OS #2 is running on the other cores. The problem of this situation is that guest OS #2 can attack the data of guest OS #1 which is in the shared main memory if guest OS #2 is running in the privileged mode. However, the data of guest OS #1 stored in the core-local memory can not be accessed by anyone else.

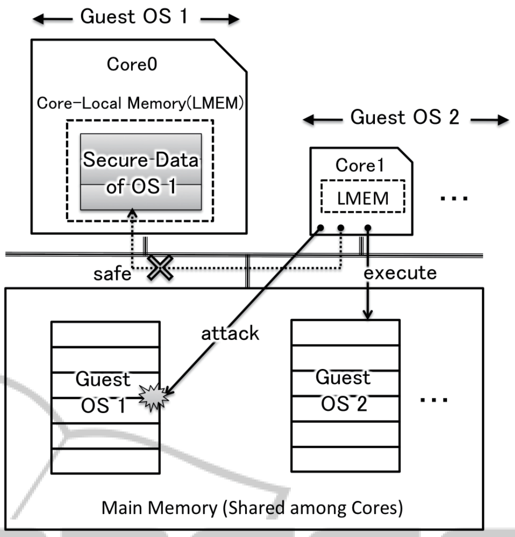

Figure 1: Behavior of core-local memory.

The main challenge of core-local memory based protection is to cope with the image size. Generally, the size of core-local memory is only a few hundred kilobytes. Therefore, it is infeasible to execute an entire modern OS and application on the core-local memory directly. To solve this problem, we implemented the checksum-based integrity management which virtually extended the capacity of core-local memory.

4.2 Checksum-based Integrity Management

In this scheme, we assume that two OSes, TOPPERS and Linux, are running on top of the virtualization layer. The goal is to prevent the attack from Linux to TOPPERS when Linux is compromised.

The kernel image of TOPPERS is allocated in main memory which is shared by each core, but linked to the virtual address space. Thus, whenever the unmapped address space of TOPPERS being accessed, the page fault events arise. Then the corresponding space in main memory is loaded to the core-local memory which is visible from the owner core only through *Secure Pager*. Secure Pager is implemented as a part of SPUMONE. Each time Secure Pager loads the data into the local memory, it first makes sure the corresponding space is not compromised, then maps virtual address to physical address within the local memory. The whole process is depicted in Fig 2.

1) First, the boot loader relocates Secure Pager into core-local memory, thus Secure Pager itself will not be directly accessed by other guest OSes. Then the boot loader relocates TOPPERS (running on core 0) and Linux (running on the other cores) into the main memory. After loading of guest OSes, Secure Pager calculates the hash values of TOPPERS per page and save the results into a table before the booting process of Linux starts. This table is also located in core-local memory, thus it can not be modified by Linux. After then, the other cores are allowed to start booting Linux.

2) Because TOPPERS are linked to the virtual address space, if TLB[1] does not contain the corresponding page table entry at the time when a page of TOPPERS is accessed, it will trigger the page fault to occur. Consequently, Secure Pager handles it as general page fault handler. We can directly handle the TLB in the case of MSRP1[2] (Ito et al., 2008) used in our implementation. The table stored with TLB entries for the mapping between virtual and physical addresses of core-local memory must not be located in the main memory.

3) During the handling of page fault, the corresponding page is copied from the main memory into core-local memory. This process is called “swap-in”. Then Secure Pager calculates the hash value of a page and compares it with the previous hash value stored in the table. If the value matches, Secure Pager loads the corresponding page table entry into TLB and completes the page fault handling. On the other hand, if the values mismatched, it indicates that certain malicious modification from Linux to the memory space of TOPPERS has occurred.

4) When core-local memory is full, Secure Pager selects a pages to be reclaimed by LRU algorithm. During the reclaim, Secure Pager re-calculates the hash value of this page and updates the table. The

[1]TLB (Table Look-aside Buffer) is the cache containing page table entries. It is designed for the CPU translating from virtual address to physical address.

[2]MSRP1 is the 4-core SH-4A multicore processor developed by Renesas Technology Corp. and Hitachi, Ltd.

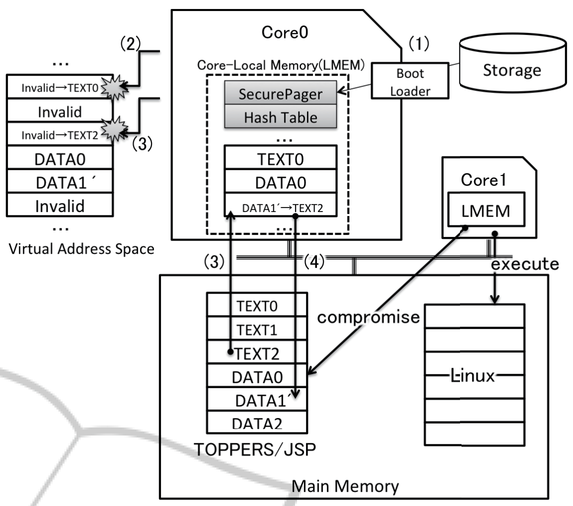

Figure 2: Checksum-based integrity management.

process is called “swap-out”. In case that the selected page is not dirty, the page-copy and re-calculation becomes unnecessary, thus Secure Pager only has to overwrite it with a new page.

Putting it all together, Secure Pager can detect the malicious modification from Linux to the main memory space of TOPPERS by checking the hash value, while all in-use pages loaded in core-local memory can not be accessed by Linux running on the other cores. As a result, the compromisation of memory space from Linux to TOPPERS is rather avoided or detected by the Secure Pager.

5 EVALUATION

This section presents the evaluation methodology and the result of Secure Pager. Because the mechanism of Secure Pager relies singularly on page fault events, significant overhead is imposed on the handling procedure of page faults and this overhead may be distributed over all aspects of system performance. Thus we focus our evaluation on two microbenchmarks: (1) page fault and (2) interrupt latency. We also observed the performance overhead against the allowed swap size.

Before going into details, we found two difficulties in our evaluation. First, we tried to find several benchmark suites on TOPPERS that can collectively display many aspects of system performance, because we can not predicate how would the overhead of page fault handling be distributed. However, we could not find such a set of benchmarks. Only few “small” benchmarks specific to certain aspects have been found and tested. For example, *Thread*

Table 2: The detail of environment setup.

Architecture	ISA	SH-4A with Multicore Extension
	CPU Frequency	400MHz
	Local Memory	128KB
Guest OS	Linux 2.6.16 TOPPERS/JSP	

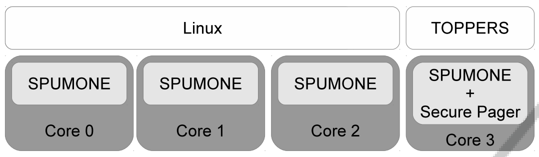

Figure 3: Configuration of guest OSes.

Metric (Express Logic, Inc.) is a benchmark which determines the realtime responsiveness by measuring scheduling overhead. Even for this case, most of them still face another problem: bad granularity for swap size configuration.

In order to observe the overhead imposed by swapping, we must eliminate the size of swap space to force the generation of swapping, even when the image of benchmark program fits into the local memory. However, most of the small benchmarks take no more than dozens of kilobytes. In this case, it would cause severe thrashing even with an one-page smaller swap space, which is also a nonrealistic environment for program execution. We address both of the difficulties by using a customized *hackbench* (Yanmin, 2008). Because the configured tasks are statically allocated and linked in TOPPERS, we can easily adjust the image size by changing the number of tasks in *hackbench*. We can also view the increase in runtime of *hackbench* as the overhead on overall performance.

5.1 Environment Setup

We took the evaluation on a multicore extension of SH-4A platform which has 4 cores, with Linux as the source of attack and TOPPERS as the protected OS. The detail of our environment is listed in Table 2. Linux is running on the core 0-2, and TOPPERS is running on the core 3. The configuration of our system is depicted in Fig 3. Despite the platform has a 128KB-sized local memory, we decided to define a certain area of main memory as the local memory rather than to use the local memory directly, thus to gain the configurability during our evaluation.

5.2 Paging Overhead

The histogram of paging overhead over 1000 samples is presented in Fig 4. This result can be roughly

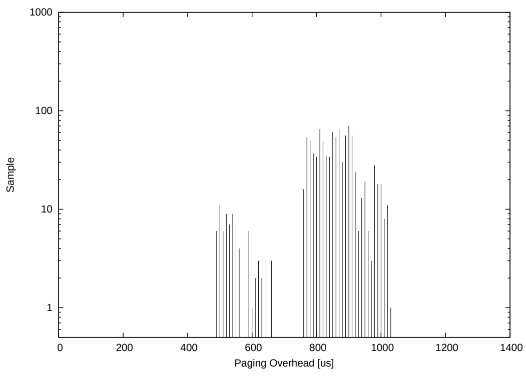

Figure 4: The result of paging overhead.

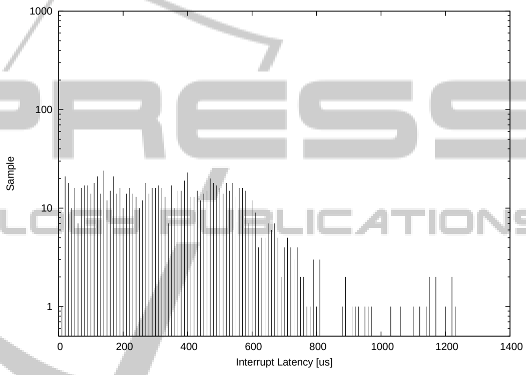

Figure 5: The result of interrupt latency.

separated into two groups, which is distinguished by whether performing write-back of dirty pages or not. Most of the cases take no more than 1ms, while rare cases (<3%) take more than 1ms to complete the swapping. The worst case does not exceed 1200μs. Because the cost of searching a page for swap-out is relatively trivial, this result do not change with the swap size configuration.

5.3 Interrupt Latency

The histogram of interrupt latency over 1000 samples is presented in Fig 5. The worst case (<3%) around 1200 -1300μs matches the result of paging overhead. Though it depends on the behavior of each interrupt events, most of the cases take no more than 800μs to perform interrupt handling. For the same reason, this result generally do not change with the swap size, except for extreme conditions when severe thrashing happens.

5.4 Result of Hackbench

Table 3 shows the result of *hackbench*. In this setup, when swap size is set to 45 pages or more,

Table 3: The result of Hackbench.

Swap Size (in # of pages)	Swap Size (in percentage)	Execution Time (μs)
45	100.00%	1048
44	97.78%	1057
43	95.56%	2704
42	93.33%	5137
41	91.11%	5271
40	88.89%	6995
39	86.67%	8933
38	84.44%	27372
37	82.22%	30290
36	80.00%	56797
35	77.78%	82853
34	75.56%	160433

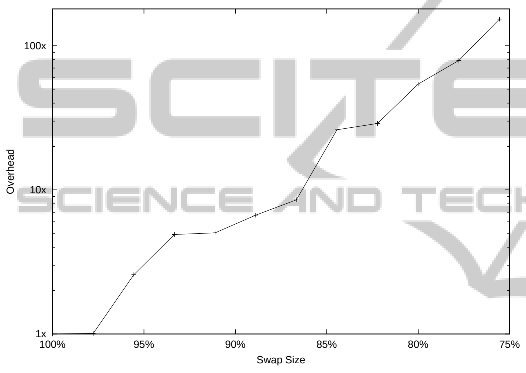

Figure 6: The result of Hackbench.

no page fault happens during the benchmark execution. Thus, the *minimal required swap space without Secure Pager* is 45 pages, and is then referred as 100%. Apart from that, the memory footprint reaches 94 pages. Apparently the execution time increased exponentially with smaller swap size. It is depicted in Fig 6.

6 DISCUSSION

6.1 Portability

Since the design of Secure Pager relies solely on the interception of page fault events, the implementation is done in the realtime hypervisor and do not require modification to Guest OSes. The only effort in adaptation of Guest OSes is to change the working address space to virtual address space, which is simply done by changes to linker script. Thus we can expect that only minimal efforts has to be done when porting Guest OSes other than TOPPERS to Secure Pager.

6.2 Performance

The evaluation results in the previous section shows the overhead is severely increased when decreasing the swap size. From the result, we can tell that the realistic ratio of swap space to *minimal required space without Secure Pager* is above 85%. However, this amount only accounts for 41% of the memory footprint. On the other hand, though it depends on the behavior of the actual running program, we can optimistically run a program on the local memory such has a far smaller size compared to the binary size, with the overhead lower than 10x. The rest of the performance overhead could be overcome easily by using processors with higher frequency. (This would introduce the cost to replace old hardware, but the effort to port old software is still eliminated.)

In terms of interrupt latency, the worst case around 1200μs is small enough for general-purpose applications. Moreover, because our evaluation is done on main memory, we can expect further speedup by running on the local memory directly, as a result of fewer access cycles. This may eventually lead to a latency around hundreds of microseconds, which should be suited for most applications that require only weak realtime responsiveness.

Throughout the results, the cost for swapping and checksum calculation accounts for almost whole of the performance overhead. Thus we could expect by utilizing hardware support for hash calculation to eliminate the cost. Further, local memory on real CPUs often accompanied with a block transfer mode which enables high-throughput page access between local memory and main memory. This kind of featured was not used in our evaluation because our goal was to determine the limitation by using minimal support from hardware. By utilizing this feature, we should be able to eliminate the cost for swapping efforts to smaller degrees, thus improve the overall performance.

7 CONCLUSIONS

In this research, we introduced Secure Pager, which has realized memory space isolation in realtime hypervisor without hardware virtualization support. Secure pager's high portability enables old software to utilize its feature with nearly no porting efforts. Moreover, it maintains the realtime responsiveness to a certain degree, which could be further improved by faster hardware. The simplicity of our approach also makes it applicable to other realtime hypervisors.

Although many realtime hypervisors have been proposed, currently there is no well-established scenario for real applications using such kind of software architecture. One application of Secure Pager is to protect a monitoring service that detects rootkits for Linux. By means of Secure Pager, it could avoid malicious access from Linux to the memory image of the monitoring service. In spite of the case, we are looking forward to have more applications to exploit this feature.

REFERENCES

Kanda, W., Yumura, Y., Kinebuchi, Y., Makijima, K. and Nakajima, T.. (2008). SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems. *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference*, vol. 1, pp. 144-151.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauery, R., Pratt, I. and Warfield, A. (2003). Xen and the Art of Virtualization. *SOSP.*

Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., Kato, K. (2009). BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. *VEE.*

Seshadri, A., Luk, M. and Qu, N. and Perrig, A. (2007). SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst.*

Advanced Micro Devices. (2011). *AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.19 edition.*

Sangorrin, D., Honda, S. and Takada, H.. (2010). Dual Operating System Architecture for Real-Time Embedded Systems. *OSPERT.*

Alves, T. and Felton, D. ARM. (2004). TrustZone: Integrated Hardware and Software Security. *Information Quarterly Volume 3.*

Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M. and Marwedel, P. (2002) Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems. *CODES.*

Held, J. (2010) Single-chip Cloud Computer: An experimental many-core processor from Intel Labs. Retrieved from: http://communities.intel.com/servlet/JiveServlet/downloadBody/5074-102-1-8131/SCC_Sympossium_Feb212010_FINAL-A.pdf. *Intel Labs Single-chip Cloud Computer Symposium.*

Shimizu, K., Hofstee, H. P. and Liberty, J. S. (2007). Cell Broadband Engine processor vault security architecture. *IBM Journal of Research and Development, Volume 51 Issue 5,* 521-528.

Kinebuchi, Y., Nakajima, T., Ganapathy, V. and Iftode, L. (2010) Core-Local Memory Assisted Protection. *Pacific Rim International Symposium on Dependable Computing, IEEE,* pp. 233-234.

TOPPRES Project. (2004). TOPPERS/JSP Kernel USER'S MANUAL. Retrieved from: http://www.ydktec.com/az/document/AZ9360SDK_TOPPERS_UM.pdf

Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K. and Kasahara, H. (2008). An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler. *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers.* IEEE International, 90-598.

Express Logic, Inc. Thread-Metric Benchmark Suite. Retrieved from: http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

Yanmin (2008). hackbench. Retrieved from: http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c