RTCSA 2006
Work in Progress Session

Editors: Timothy Bourke and Stefan M. Petters
Work-in-Progress-Chair: Liu Xiang, Peking University, China

National ICT Australia
223 Anzac Parade
Kensington NSW 2052
Australia
{firstname.lastname}@nicta.com.au

Technical Report August 2006

ISSN 1833-9646

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

Contents

A Fast System Call Implementation for Embedded Systems

Shi-Wu Lo
Department of Computer Science and Information Engineering
National Cheng-Chung University
shiwulo@cs.ccu.edu.tw

Abstract

User mode program usually requires the assistance from hardware in order to execute a mode change before using system service. This also allows user mode program to gain complete control of hardware. In this paper, new instructions are proposed; the OS may then take advantage of these instructions to more effectively use the system service, while reducing the kernel size and adding to the flexibility in design.

Introduction

System call acts as a communication interface between the *Operation System* (OS) and *User Mode Program* (UMP). To guarantee that all hardware resources are under the control of OS, the processor must have 2 or more execution modes (i.e. kernel mode and user mode). In addition, mode changes (e.g. user mode => kernel mode) must be executed such that system security is not harmed in any way.

Currently, most OS's and CPUs use software interrupts to complete mode changes. Since all system calls have the same entry point, kernels must use certain types of dispatch mechanisms (e.g. table lookup) to allow the mapping between system calls and *Kernel Service Routines* (KSR). Such mechanisms make the pre-processing mechanisms of system calls fairly complex. Thus, in this paper we propose a new method called *Fast System Call* (FSC) to increase the efficiency of system calls.

Take as an example Linux's system calls operating on a Pentium processor. Figure 1 shows that 5 steps are required to complete a system call: 1. UMP generates a software interrupt (i.e. int 0x80); 2. processor loads the content at location 0x80 in the *Interrupt Descript Table* (IDT) into the *Program Counter* (PC); 3. when the new PC is loaded, the OS will call out to different KSRs based on different value of AX registers; 4. returning from the called KSR; 5. returning from kernel.

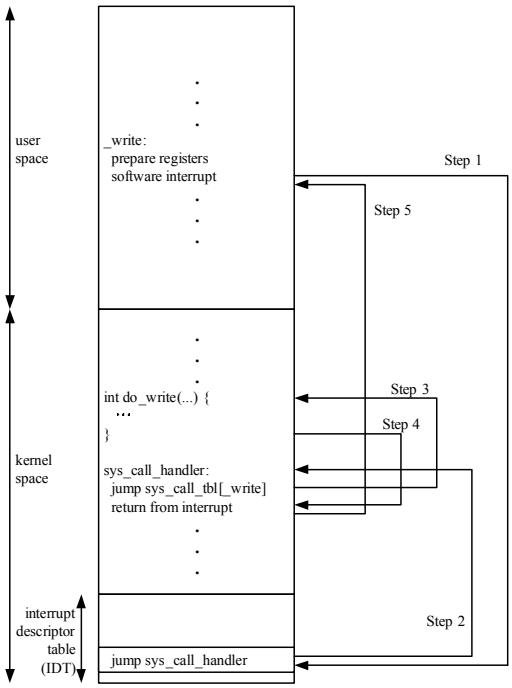


Fig. 1 Linux's system call on a Pentium processor

Since interrupt-based system calls must call out to the correct KSRs using software's dispatch mechanisms (Step 3), the execution of those applications with more stringent time requirements must be done inside kernels (e.g. Linux's kernel module). Doing so raises efficiency, but lowers system stability. In this paper, we are hoping to increase system call efficiency by performing minor CPU modifications.

The Fast System Call Implementation

The FSC utilizes a cipher register to limit the entry points of kernel. This way the UMP can directly call out to the particular KSRs. The proposed method is shown in the figure below:

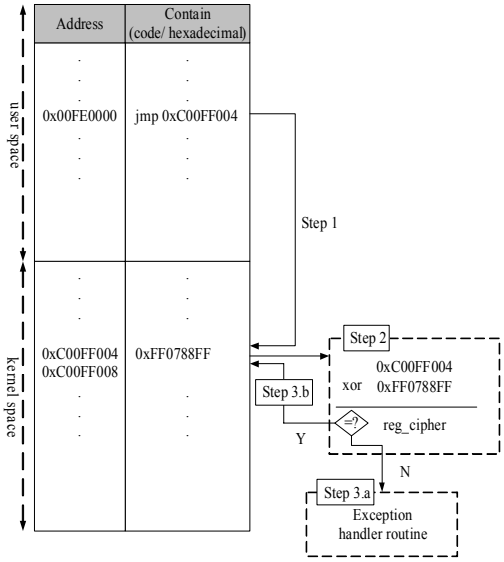


Fig.2 The new method

As shown in Figure 2, we have omitted some details for simplicity purpose. For example, because immediate addressing cannot address all address spaces, the destination addresses may first need to be copied to registers, before a jmp instruction can be performed.

When UMP is executing the instruction (i.e. jmp 0x C00FF004) stored in memory location 0x00FE0000, it will detect that the next instruction (at 0x C00FF004 in Figure 2) to be executed is located in kernel space. Thus the CPU will handle the instruction/word from 0xC00FF004 in a special way. First, the instruction/word from 0xC00FF004 (i.e. the instruction/word represented by 0xFF0788FF) will be Exclusive ORed with the memory address of the instruction/word (i.e. 0xC00FF004). (This is shown in Step 2 found in Figure 2.) The result of this Exclusive OR

operation is then compared to the cipher register (the reg_cipher as shown in Figure 2). If the comparison shows that the two are equal, then the CPU can immediately switch over to kernel mode and continue the execution (Step 3.b in Figure 2). On the other hand, if the comparison shows that the two are not equal, then system generates an exception, and an Exception Handler Routine (EHR) is executed instead (Step 3.a in Figure 2).

This approach primarily serves 2 purposes:

1. Simplify the process flows followed by the OS when processing system calls.
2. Given the premise that the system security is not to be harmed, FSC allows the kernel to simultaneously store code and data on the same page. As such, the kernel becomes smaller and more efficient.

As described earlier, most kernels have a single entry point. When this is the case, the processing steps for system calls become fairly complex. (Please refer to the explanations in the Introduction section.) In contrast, the new method directly executes the corresponding KSR, resulting in less overhead.

The Proposes of Encryption

To make the kernel even smaller and more efficient, we must encrypt KSR's first instruction. If encryption is not applied to protect KSR's entry point, the kernel must then divide up all pages strictly into data pages and code pages. Then, the XD bit (execute disable bit) of data pages can be set to prevent these pages from executing. The method (FSC) proposed by this paper allows the placement of code segment and data segment on the same page, which helps lower internal fragmentation and thereby enhances the utilization rate of kernel memory. On the other hand, since code segment and data segment are placed side by side, the compiler may use as much PC relative addressing as possible to raise the particular module's efficiency and size (as shown in Figure 3).

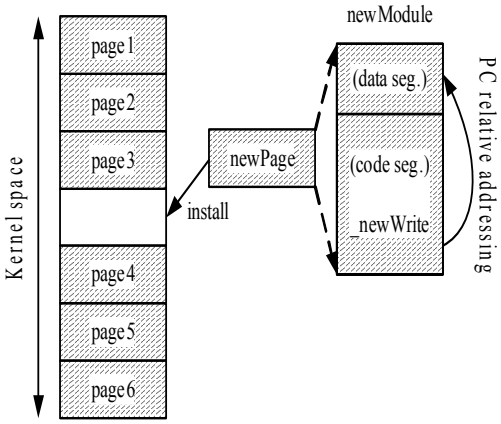


Fig.3 The kernel modules can be inserted into the kernel more easily and efficiently

For example, without FSC and dividing up all pages into data pages and cold pages, a malicious program may use a

system call "write" to disguise itself as data and write itself into a data buffer within the kernel. If this malicious program learns the location of this data buffer from studying the code of Linux (assuming the location is 0xCF00044), it may use the instruction jmp 0xCF00044 to execute the illegal code. However, because UMP has no way of learning the value inside reg_cipher[1], the malicious program will be unable to fill in the correct value into 0xCF00044. As such the program that begins at 0xCF00044 cannot be certified, and the CPU would automatically begin the execution of the corresponding EHR. Within the EHR, the OS will terminate the malicious program and report an error (Step 3.a in Figure 2).

Related Work

Most CPUs and OS's (such as SI from the ARM[1] series, and the int 0x80 from the Intel x86 series [2] running on Linux) adopt a system call design that is software interrupt based. In addition, some processors (such as Intel IA-64[3] and IA-32[4]) provide call gate mechanism to raise the processing efficiency of system calls. Call gate is located on a special page; through it the UMP can directly execute kernel service. The function of call gate is similar to *interrupt vector table* (IVT). Call gates are located on independent pages/segments because call gates should have special permission. By having the call gate call out to KSR, the CPU can automatically switch over from user mode to kernel mode. Since the call gates are located on independent pages/segments, it consumes more memories in contrast to FSC. Also, in contrast to the 2 existing methods (call gates and interrupt based system call), when the system adds in a new module which is providing a system call, the FSC requires no special processing to export the provided system call. (As shown in Figure 3, the new system call within newModule is called _newWrite. When a super user adds this module into kernel, kernel just adds ciphertext in front of _newWrite, enabling the UMP to use the new system call.)

The method used by language-based OS represents an extreme case, which totally relies on software's dynamic checks to limit the usage of pointer and hardware. An example of such products is the JavaOS [5], whose main disadvantage is poor efficiency during execution. The other extreme case is an over-reliance on the protection provided by hardware, which extends full protection to every component of the OS during execution. The main disadvantage here is higher hardware cost [6]. The method proposed in this paper is generally similar to the methods used by CPUs and OS's in general. As such, both software and hardware require only minor modifications. But

[1] This value should be assigned by a random function while booting the OS.

the newly proposed method is more superior in terms of efficiency and size of the kernel.

Conclusion & Future Work

System call is a communication interface between UMP and kernel, and as such the efficiency of system call is very much related to the efficiency of the OS. When the efficiency of system call falters, a programmer may try to move all programs into kernel. But doing so may seriously damage the system and bring harm to stability. This paper proposes a simple mechanism which makes the processing of system call more efficient. In addition, it allows compiler to make full use of PC relative addressing, thus making the modules within kernels smaller and more efficient.

For security reasons, UMP and kernel often have to use different stacks. Switching stack degrades the locality stored in memory, while parameters need to be transmitted between UMP and kernel using special method. This makes compiler optimization harder when transmitting parameters. In the future, we will measure the improved efficiency of fast system calls over simulator, and will perform in-depth research into the issue of switching stack.

Reference

[1] David Seal. "ARM Architecture Reference Manual," Addison-Wesley, London, UK, 2000.

[2] M. Beck, H. B ohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. "Linux Kernel Internals," Addison-Wesley, 2nd edition, 1998.

[3] Intel Corp. "IA-32 Intel® Architecture Software Developer's Manuals," http://www.intel.com/design/pentium4/manuals/index_new.htm.

[4] Intel Corp. "Intel Itanium Architecture Software Developer's Manual," http://www.intel.com/design/itanium/manuals/iiasdmanual.htm.

[5] P. Madany, et. al. "JavaOS: A Standalone Java Environment," White Paper, Sun Microsystems, May 1996.

[6] Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traege. "Efficient and Safe Execution of User-Level Code in the Kernel," in IPDPS'05, pp. 221.

An Efficient Model-Mapping-Schema-Based Approach for Real-Time Access to XML Database Systems

Chih-Chiang Lee, Jun Wu†, Chih-Wen Hsueh and Tei-Wei Kuo

Department of Computer Science and Information Engineering
Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106, ROC
{p89005,cwhsueh,ktw}@csie.ntu.edu.tw

†Department of Information Technology
National Pingtung Institute of Commerce
Pingtung, Taiwan 900, ROC
junwu@npic.edu.tw

Abstract

In this paper, we propose a new model-mapping schema, called XRoute, to translate and manipulate XML documents in a relational database. This research is motivated by the potential performance problem of model-mapping-schema-based approaches for providing real-time access to XML data. We propose a Dewey-based method to encode the path information of XML data as ID's of each element or attribute to reduce significant join cost. A translation algorithm is proposed to translate XML documents into records in a relational database. Query processing algorithms are also proposed to translate XML queries into SQL commands. The capability of the proposed algorithms were verified by a series of simulation experiments based on the XML benchmark [1], for which we have some encouraging experimental results.

1 Introduction

Extensible markup language (XML) is widely accepted as a universal standard for information exchange because of its simple and flexible text format. How to design an efficient XML storage system for providing real-time access of XML documents is an active research topic in recent years. With the popularity of *relational database systems* (RDBS), many researchers and vendors (including Oracle, Sybase, IBM, and Microsoft) have proposed their approaches based on RDBS [2, 3, 4, 5, 6, 7, 8, 9] to store and manipulate XML data as relational tables. These relational-database-based approaches need schema definitions to translate XML documents into relational tables. Such schema definitions can be classified into two categories[7]: *structure-mapping schema* and *model-mapping schema*.

Under the structure-mapping schema [2, 3, 5, 6], an XML document is translated into relational tables based on its structure (i.e., *document type descriptor* (DTD)). As a result, each XML document might has different database schemas. It creates additional complexity for managing different structured XML documents with different logical and physical designs of relational database systems. On the contrary, the model-mapping schema [4, 5, 7, 8, 9] provides the same set of relation definitions for any XML document. That is, every XML document has the same database schema. This approach can support any sophisticated applications and well-formed XML documents, even though they do not have DTD's. The Edge [4], Monet [5], XRel [7] and XParent [8, 9] are examples schemas based on the model-mapping schema. In particular, Jiang et al. [8, 9] shows that the performance of the XParent schema is better than other model-mapping schemas. Although different model-mapping schemas have different relation definitions for XML documents, every model-mapping schema needs to assign an ID to each element or attribute of an XML document. The ID's of elements and attributes usually serve as the primary key or the foreign key of the relations in an XML document so that a large number of join operations might be needed to locate elements and attributes in a query. Such a significant join cost is a potential performance problem for providing real-time access of XML data.

This paper explores the performance issues of XML data access over relational databases. Based on the well-known *Dewey decimal classification system* [10], we encode the structure information of XML data as ID's of elements and attributes in XML documents. With the encoded structure information, we propose a new model-mapping schema, called *XRoute*, and a translation algorithm to translate XML documents into records in relational databases. We also propose query processing algorithms to translate XML queries into SQL commands. Related work on the encoding of the structure information of XML data was done

by Tatarinov et al. [11], in which the Edge schema with arbitrary unique ID's and Dewey-based ID's were evaluated[1]. We shall show later that a large number of joins could be avoided for many kinds of queries. The capability of the proposed methodology and algorithms in this paper are verified by a series of simulation experiments under different patterns of XML documents generated by the XML Benchmark Project [1], for which we have some encouraging experimental results.

The rest of this paper is organized as follows: Section 2 introduces related work on model-mapping schema and discusses the potential performance problem. Section 3 proposes algorithms for data translation and manipulation. Section 4 reports our work on simulation experiments. Section 5 is the conclusion and future work.

2 Related Work

There are some well-known model-mapping schemas are often referenced in the research of XML storage systems, such as Edge [4], Monet [5], XRel [7], and XParent [8, 9]. Because of space limit, we only introduce the XParent schema [8, 9]. The database schema of XParent is as follows:

LabelPath(ID,Len,Path)
DataPath(Pid,Cid)
Element(PathID,Did,Ordinal)
Data(PathID,Did,Ordinal,Value)

Table LabelPath stores the information of label-paths of an XML data graph. The attributes ID and Len denote the unique ID and the length (the number of edges in the label-path) of each label-path, respectively. The attribute Path denotes the name of the corresponding label-path which is a sequence of node names in the label-path. Table DataPath stores the information of parent-child relationships of an XML data graph. The attributes Pid and Cid denote the node number of the corresponding parent node and child node of an edge, respectively. Table Data stores the information of the nodes of the XML data graph if the corresponding elements or attributes in an XML document have a value. The attributes PathID and Did denote the ID of label-path (i.e., the foreign key of the ID in table LabelPath) and the node number of the node, respectively. The attribute Ordinal denotes the ordinal number of the node among its sibling-nodes with the same name. The attribute Value denotes the value of the node, where the value of a node is the value of the corresponding element or attribute in the XML document. Table Element is very similar with Table Data, except that Table Element stores the information of all nodes in an XML data graph, and Table Element does not store the value of each node. Figures 1 and 2 show an example XML document and its corresponding data graph [12]. Under the XParent schema, there are four tables as shown in Figure 3.

[1]Different form the past work proposed by Tatarinov et al. [11], we did not directly apply our ID encoding method to some existing model-mapping schema. It is clear that a new model-mapping schema is needed, when structure information of XML data are already stored in ID's.

```
<bib>
    <video year="2002">
        <title>Harry Potter and the Chamber of Secrets</title>
        <starring>
            <lastname>Radcliffe</lastname>
            <firstname>Daniel</firstname>
        </starring>
        <distributor>Warner Brothers</distributor>
        <length>2hrs. 41min.</length>
    </video>
    <video year="2002">
        <title>The Santa Clause</title>
        <starring>
            <lastname>Allen</lastname>
            <firstname>Tim</firstname>
        </starring>
        <distributor>Walt Disney</distributor>
        <length>1hrs. 45min.</length>
    </video>
    <video year="2002">
        <title>Comedy</title>
        <starring>
            <lastname>Cube</lastname>
            <firstname>Ice</firstname>
        </starring>
        <distributor>New Line Cinema</distributor>
        <length>1hrs. 25min.</length>
    </video>
</bib>
```

Figure 1: An example XML document.

The XParent needs a large number of joins to locate all of the edges that correspond to an element or attribute that appears in a given query. Because join operations are very costly, the number of joins for a query is proportional to the processing time of the query. Such an observation motivates this research. In this paper, a new model-mapping schema will be proposed. Our objective is to reduce the number of joins to locate edges corresponding to elements and attributes in a query such that better performance can be obtained for providing real-time access to XML data.

3 XRoute - Translation and Manipulation of XML Documents

In this section, we propose a new model-mapping schema, called XRoute, and algorithms to translate and manipulate XML documents over relational database systems.

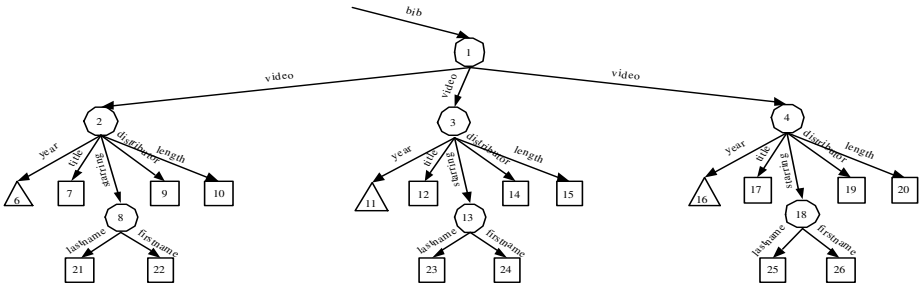


Figure 2: The corresponding XML data graph of an XML document in Figure 1.

ID	Len	Path
1	1	/bib
2	2	/bib/video
3	3	/bib/video/year
4	3	/bib/video/title
5	3	/bib/video/starring
6	4	/bib/video/starring/lastname
7	4	/bib/video/starring/firstname
8	3	/bib/video/distributor
9	3	/bib/video/length

Table LabelPath

PathID	Did	Ordinal	Value
3	6	1	2002
4	7	1	Harry Potter and the Chamber of Secrets
8	9	1	Warner Brothers
9	10	1	2 hrs. 41 min.
3	11	1	2002
4	12	1	The Santa Clause
8	14	1	Walt Disney
9	15	1	1 hr. 45 min.
3	16	1	2002
4	17	1	Comedy
8	19	1	New Line Cinema
9	20	1	1 hr. 25 min.
6	21	1	Radcliffe
7	22	1	Daniel
6	23	1	Allen
7	24	1	Tim
6	25	1	Cube
7	26	1	Ice

Table Data

PathID	Ordinal	Did
1	1	1
2	1	2
2	2	3
2	3	4
3	1	6
4	1	7
5	1	8
8	1	9
9	1	10
3	1	11
4	1	12
5	1	13
8	1	14
9	1	15
3	1	16
4	1	17
5	1	18
8	1	19
9	1	20
6	1	21
7	1	22
6	1	23
7	1	24
6	1	25
7	1	26

Table Element

Pid	Cid
1	2
1	3
1	4
2	6
2	7
2	8
2	9
2	10
3	11
3	12
3	13
3	14
3	15
4	16
4	17
4	18
4	19
4	20
8	21
8	22
13	23
13	24
18	25
18	26

Table DataPath

Figure 3: The four tables of the XML data graph in Figure 2 under the XParent schema.

3.1 XRoute Schema

The database schema of XRoute is as follows:

Path(PathID, LabelPath, Type)
Node(NodeID, PathID, NodeValue)

Table Path stores the information of label-paths of an XML data graph, where the label-path of a node is a sequence of node names from the root to the node. The attribute PathID denotes the unique ID of each label-path. The attribute LabelPath denotes the name of the corresponding label-path which is a sequence of node names in the label-path. The attribute Type indicates whether the last node in the corresponding label-path is an element (i.e., Type=E) or an attribute (i.e., Type=A). Table Node stores the information of elements and attributes of an XML document. The attribute NodeID denotes the unique ID of each node which is encoded based on the Dewey decimal classification system [10]. The Dewey-based ID encoding method is as follows: The NodeID of a node is a combination of the NodeID of its parent node and the ordinal number among nodes which connected by the same parent node. Note that the NodeID of the root node of an XML document is defined as '0'. Suppose that a node $Node_c$ is the nth children of the node $Node_p$ (i.e., the ordinal number of $Node_c$ is n), and the NodeID of a node $Node_i$ is denote as NodeID($Node_i$). By the Dewey-based ID encoding method, the NodeID of $Node_c$ is represented as NodeID($Node_p$) $+ : +(n-1)$, where : is a separate-character and the value of $n-1$ could be represented as a character[2]. As a result, the attribute NodeID stores the information of ancestors of each node. The attribute PathID denotes the ID of the label-path (i.e., the foreign key of the PathID in Table Path) of each node. The attribute NodeValue denotes the value of the node, where the value of a node is the value of the corresponding element or attribute in the XML document. Figure 4 shows the corresponding XML data graph with the encoded NodeID's of the XML document shown in Figure 1.

Our proposed encoding method encodes the information of ancestors of each element or attribute of an XML document, the decoding of the NodeID completes a query instead of the large number of joins that needed by other model-mapping schemas. Because the cost of join operations are higher than the decoding of the NodeID, reduce the number of joins could significantly improve on the performance of an XML storage system.

3.2 Translation Methodology and Algorithm

The proposed translation algorithm, called *XML2RDB*, is as follows[3]:

[2]When the maximum number of ordinal number is greater than 256, then the wide character (e.g., the Unicode) should be adopted.

[3]Where the $Parent(node_i)$ and the $Ordinal(node_i)$ are functions that return the parent-node and the ordinal of the node

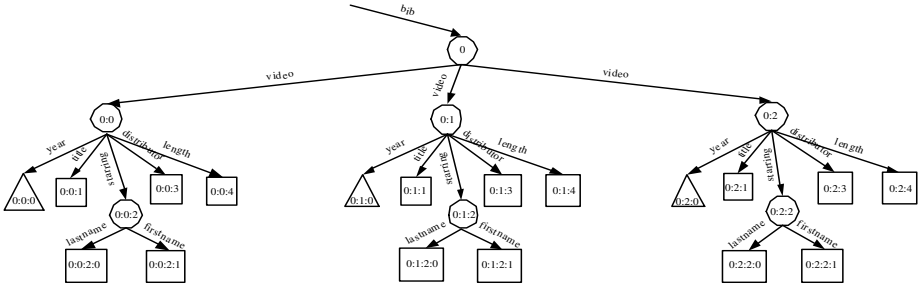


Figure 4: The corresponding XML data graph with encoded NodeID's of the XML document in Figure 1.

Algorithm ***XML2RDB***
Input: an XML document
Output: generate and execute a series of SQL commands
begin
1: $pid := 0$
2: $DOMTree :=$ the DOM tree of the input XML document
3: generate and execute SQL commands that create the Table Node and the Table Path
4: **for each** $node_i \in DOMTree$ **do**
5: $nodeName_i :=$ the name of $node_i$
6: $labelPath_i := labelPath_{Parent(node_i)} + '/' + nodeName_i$
7: **if** $labelPath_i$ not exists in the Table Path **then**
8: generate and execute a SQL command that insert a tuple (pid ++, $labelPath_i$, 'E') into the Table Path
9: **end if**
10: $pathID_i := GetPathID(lablePath_i)$
11: $nodeID_i := nodeID_{Parent(node_i)} + ' : ' + Ordinal(node_i)$
12: $nodeValue_i :=$ the value of $node_i$
13: generate and execute a SQL command that insert a tuple ($nodeID_i$, $pathID_i$, $nodeValue_i$) into the Table Node
14: **for each** $attribute_j \in node_i$ **do**
15: $attributeName_j :=$ the name of $attribute_j$
16: $labelPath_j := labelPath_{Parent(node_i)} + ' / ' + attributeName_j$
17: **if** $labelPath_j$ not exists in the Table Path **then**
18: generate and execute a SQL command that insert a tuple (pid++, $labelPath_j$, 'A') into the Table Path
19: **end if**
20: $pathID_j := GetPathID(\ labelPath_j)$
21: $nodeID_j := nodeID_i + ' : ' + Ordinal(attribute_j)$
22: $nodeValue_j :=$ the value of $attribute_j$
23: generate and execute a SQL command that insert a tuple ($nodeID_j$,$pathID_j$, $nodeValue_j$) into the Table Node
24: **end for**
25: **end for**
end.

Let n be the number of elements and attributes in an XML document. It is easy to see that the time complexity of *XML2RDB* is $O(n)$. Under the XRoute schema, Table Node and Table Path of the XML data graph in Figure 2 are shown in Figure 5.

3.3 Manipulation of XML Documents

In this paper, we assume that user queries are represented in XQuery [12] commands. To manipulate XML data, it is needed to translate XQuery commands into a sequence of corresponding SQL commands. Figure 6 is an XQuery example that gets the title of videos published by New Line Cinema in 2002 from the XML document in Figure 1.

$node_i$, respectively. The $GetPathID(labelPath_j)$ is a function that return the PathID of the $labelPath_j$ from Table Path.

NodeID	PathID	NodeValue
0	1	
0:0	2	
0:1	2	
0:2	2	
0:0:0	3	2002
0:0:1	4	Harry Potter and the Chamber of Secrets
0:0:2	5	
0:0:3	8	Warner Brothers
0:0:4	9	2 hrs. 41 min.
0:1:0	3	2002
0:1:1	4	The Santa Clause
0:1:2	5	
0:1:3	8	Walt Disney
0:1:4	9	1 hr. 45 min.
0:2:0	3	2002
0:2:1	4	Comedy
0:2:2	5	
0:2:3	8	New Line Cinema
0:2:4	9	1 hr. 25 min.
0:0:2:0	6	Radcliffe
0:0:2:1	7	Daniel
0:1:2:0	6	Allen
0:1:2:1	7	Tim
0:2:2:0	6	Cube
0:2:2:1	7	Ice

Table Node

PathID	LabelPath	NodeType
1	/bib	E
2	/bib/video	E
3	/bib/video/year	A
4	/bib/video/title	E
5	/bib/video/starring	E
6	/bib/video/starring/lastname	E
7	/bib/video/starring/firstname	E
8	/bib/video/distributor	E
9	/bib/video/length	E

Table Path

Figure 5: Table Node and Table Path of the XML data grpah in Figure 2 under the XRoute schema.

The proposed algorithm is as follows: We parse the XQuery command and determine corresponding name of Table Node and Table Path (e.g., video_Node and video_Path of Figure 6) according to the filename of the XML document specified in the FOR clause. According to names of tables (Table Node and Table Path), variables in FOR and LET clauses, and the condition-expression in the WHERE clause, the corresponding SELECT command of SQL is generated. It is easy to see that the time complexity of this algorithm is $O(n)$, where n is the length of an XQuery command. Figure 7 shows a SQL command with 5 equijoins and 5 selections corresponding to the XQuery command in Figure 6. Where the PARENT_NODEID() is a function that returns the NodeID

```
FOR $x IN document("videoxml")/bib/video
LET $y=$x/distributor
WHERE $y="New Line Cinema"
  AND $x/@year="2002"
RETURN $x/title
```

Figure 6: An example XQuery command.

```
SELECT N2.NodeValue
FROM video_Node N0, video_Path P0,
     video_Node N1, video_Path P1,
     video_Node N2, video_Path P2
WHERE P0.LabelPath='/bib/video/distributor'
      AND N0.PathID = P0.PathID
      AND N0.NodeValue='New Line Cinema'
      AND P1.LabelPath='/bib/video/year'
      AND N1.PathID = P1.PathID
      AND N1.NodeValue='2002'
      AND P2.LabelPath='/bib/video/title'
      AND P2.PathID=N2.PathID
      AND PARENT_NODEID(N2.NodeID)=PARENT_NODEID(N0.NodeID)
      AND PARENT_NODEID(N2.NodeID)=PARENT_NODEID(N1.NodeID);
```

Figure 7: The corresponding SQL command of the XQuery command in Figure 6 under the XRoute schema.

of the parent-node of a given node. For comparison, Figure 8 shows a SQL command under the XParent schema corresponding to the XQuery command in Figure 6. When the XParent schema is adopted, 8 equi-joins and 5 selections are used.

4 Performance Evaluation

Although researchers have been proposed various model-mapping schemas, we only compare our work with the XParent schema. It is because Jiang et al. [8] shows that the performance of the XParent schema is better than other model-mapping schemas.

4.1 Metrics and Data Sets

The primary performance metric is the response time of an operation, referred to as $ResponseTime$. Other interested metrics are $CPUTime$ and $NumIO$, where the $CPUTime$ and the $NumIO$ are the CPU computation time of an operation and the number of I/O activities of an operation, respectively.

The experiments are conducted on a PC with one *Pentium* III 700 MHz processor and 512MB RAM, and the relational database system is Oracle8*i*. The testing data are generated by an XML document generator, called *xmlgen*, which is provided by the XML benchmark project (XMLBENCH) [1]. The size of XML documents generated by the *xmlgen* are 1MB, 10MB, and 50MB. In our experiments, we used several

```
SELECT d2.Value
FROM LabelPath lp0, Data d0, DataPath dp0,
     LabelPath lp1, Data d1, DataPath dp1,
     LabelPath lp2, Data d2, DataPath dp2
WHERE lp0.Path = '/bib/video/distributor'
      AND d0.PathID = lp0.Id
      AND d0.Value = 'New Line Cinema'
      AND lp1.Path='/bib/video/year'
      AND d1.PathID=lp1.Id
      AND d1.Value='2002'
      AND lp2.Path = '/bib/video/title'
      AND d2.PathID = lp2.Id
      AND d0.Did = dp0.Cid
      AND d1.Did = dp1.Cid
      AND d2.Did=dp2.Cid
      AND dp0.Pid = dp2.Pid
      AND dp1.Pid =dp2.Pid;
```

Figure 8: The corresponding SQL command of the XQuery command in Figure 6 under the XParent schema.

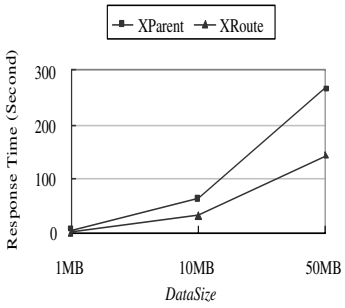


Figure 9: The average response time for translating XML documents into records in the relational database under the XParent schmea and the XRoute schema, when $DataSize$ were 1MB, 10MB, and 50MB.

benchmark query templates which were also provided by the XMLBENCH.

4.2 Experimental Results

Figure 9 shows the average $ResponseTime$ for translating XML documents into records in an RDBMS under the XParent and the XRoute schemas, when $DataSize$ were 1MB, 10MB, and 50MB. Where the $DataSize$ are the size of XML documents generated by the *xmlgen*. It was shown that the XRoute schema greatly outperforms the XParent schema. In particular, the XRoute schema reduced almost 50% time for translating, when $DataSize$ was 50MB. It is because the XParent schema needs more number of tables than the XRoute schema.

Figure 10 shows the $ResponseTime$, $CPUTime$, and $NumIO$ of query processing, when $DataSize$ is 1MB. All queries under the XRoute schema outperform that under the XParent schema except Q6. It is because the number of joins needed by the XRoute

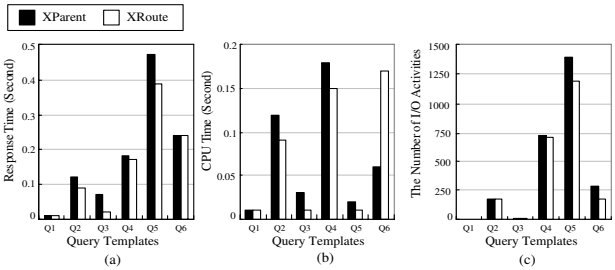


Figure 10: The response time, CPU time, and the number of I/O activities of query processing under the XParent schema and the XRoute schema, when $DataSize$ was 1MB.

schema is less than that under the XParent schema. The $CPUTime$ of Q6 under the XRoute schema takes longer than that under the XParent schema, but the XRoute schema takes less number of I/O activities. Because the number of joins for query processing and the number of tables needed by the XRoute schema are both less than that under the XParent schema, the $NumIO$ of query processing under the XRoute schema outperforms than that under the XParent schema. When $DataSize$ are 10MB and 50MB, the simulation results are similar to that in Figure 10.

The above experimental results show that the XRoute schema outperforms the XParent schema. Furthermore, the space overhead due to the storing of path information in node ID's must be paid for the improvement of the system performance. In the experiments, the space overhead was measured. When the total size of XML documents under evaluation was 50MB, the space overhead of the XRoute schema was 8.76% more than that under the XParent schema. However, the average response time of query processing under the XRoute schema was improved by 22.53%, compared to that under the XParent schema.

5 Conclusion

In this paper, we explores the performance issues on the access of XML documents over relational databases. A new model-mapping schema, called XRoute, is proposed to manipulate XML documents over relational databases. It was shown that a significant number of joins could be avoided for many kinds of queries. In addition, the required time for the storing of XML documents into databases could be reduced because of a less amount of data being stored.

For the future research, we shall further explore the possibility in reducing the CPU time consumption for queries of alphabetically ordered lists of items. We will evaluate the reorganization of the database in facing different kinds of data-insertion,-deletion,and-updating patterns under different types of XML documents. Better index structures for XML documents are also worthy of further study.

References

[1] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse, "The XML benchmark project," Technical report INS-R0103, CWI, The Netherlands. April 30, 2001.

[2] J. Kyung-Soo, "A Design of Middleware Components for the Connection between XML and RDB," In Proceeding of the IEEE ISIE, Pusan, KOREA, pp.1753-1756, 2001.

[3] R. Bourret, C. Bornhovd and A. Buchmann, "A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases," In Proceeding of the 2nd WECWIS, Milpitas, California, pp.134-143, June 8-9, 2000.

[4] D. Florescu and D. Kossman, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database," INRIA Research Report No.3680, Rocquencourt, France, May 1999.

[5] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer and Florian Waas, "Efficient Relational Storage and Retrieval of XML Documents," In Proceeding of the 3rd WebDB, Dallas, Texas, May 18-19, 2000.

[6] Ismailcem Budak Arpinar, John Miller, and Amit P. Sheth, "An Efficient Data Extraction and Storage Utility for XML Documents," In Proceedings of the 39th ACMSE, Athens, Georgia, pp. 293-295, March 2001.

[7] YoshiKawa, M. and Amagasa, T., "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases," ACM Transactions on Internet Technology (TOIT), Vol. 1, No. 1, pp.110-141, August 2001.

[8] H. Jiang, H. Lu,W.Wang, and J. X. Yu., "Path Materialization Revisited: An Efficient Storage Model for XML Data," In Procedings of ADC, 2002.

[9] Haifeng Jiang, Hongjun Lu, Wei Wang, Jeffrey Xu Yu, "XParent: An Efficient RDBMS-Based XML Database System," In Proceedings of the 18th ICDE, San Jose, California, February 26 - March 1, 2002. pp. 335-336.

[10] Lois Mai Chan, "Dewey Decimal Classification: Principles and Application," OCLC, ISBN: 999090586X, October 2003.

[11] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang, "Storing and Querying Ordered XML Using a Relational Database System," ACM SIGMOD, June 4-6, 2002, Madison, Wisconsin, USA.

[12] World Wide Web Consortium (W3C), "XQuery 1.0: An XML Query Language," http://www.w3.org/TR/2003/WD-xquery-20030502, 2003.

Genetic-based Approach for Scheduling, Allocation, and Mapping for HW/SW Co-Design with Dynamic Allocation Threshold

Chun-Nan Chou Yi-An Chen Chi-Sheng Shih
National Taiwan University, Taipei, Taiwan 106, ROC
{b91069,cshih}@csie.ntu.edu.tw

Abstract

We develop a genetic-based approach for system-level architecture synthesis for scheduling, allocation, and mapping. Traditional design practices lead to over allocating resources for embedded SoC and, hence, high manufacture cost. The approach determines the scheduling, allocation, and mapping for the system at every step to find the near optimal solution. Unlike other genetic approach for parallel processing in the literature, a new chromosome encoding is designed to avoid generating invalid system architecture and, hence, reduce the number of generations to obtain near optimal solutions. In addition, a dynamic allocation probability is designed to reduce cost and run-time overhead. We evaluate our approach with extensive simulations and compare our performance with other approaches in the literature. The results show that our approach increases schedulability up to 80% and reduces the cost up to 60% for the tested workload, comparing to earlier approaches. Furthermore, its run-time overhead is still acceptable for complex system design.

I. Introduction

With the advance technology on VLSI design, it is now affordable to customize every embedded system with off-the-shelf microprocessors and pre-designed application-specific integrated circuits (ASIC's). Hence, every embedded system can have its own system architecture design to meet its performance requirement and to minimize the manufacture cost. However, without proper system-level synthesis tools, the engineers mostly reply on their experience on designing past systems to design new embedded systems. To assure that the system can meet its performance requirement, the designers usually over allocate the resources including hardware components and slack times to meet its performance requirements. One of the reasons is that the solution space to find an optimal architecture design exponentially grows as the number of functionalities for the system increases.

Many modern embedded systems such as multi-media applications require the use of micro-processors and application-specific integrated circuits (ASIC's) on the same platform to meet their performance requirements. Let's take the video phone as an example. H.264 encoding/decoding algorithms and MPEG1 algorithm have to been implemented on the same device to support video call and music playback. From a technical perspective, H.264 requires extensive processing that must be performed in real-time to be useful. While today's CPUs continue to keep pace with Moore's Law, traditional general processors do not have architectures that are well-suited for video processing. Clearly, this calls for hardware acceleration. Figure 1 shows the data flow for encoding and decoding flows for H.264 and MPEG1 on video phones. In this example, both H.264 decoding

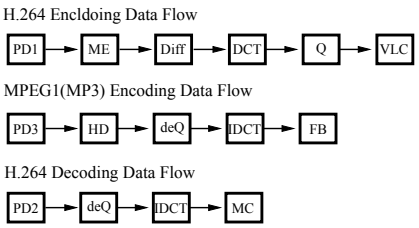


Fig. 1. Video Phone Application SoC

flow and MP3 decoding flow require IDCT and deQ. To reduce manufacture cost, it is very likely

to let H.264 decoding and MP3 decoding task flows share the processing elements designed for IDCT and deQ. Unlike general microprocessors, such processing elements (or called IP) cannot be interrupted during the execution of one task. Otherwise, the result of the task is lost and the task fails. The system may need to start the task flow from the beginning. Consequently, the task may fail to meet its performance requirement such as jitter requirement for H.264 decoding.

The challenge for system-level design is to determine what processing elements should be used in the design and how the selected processing elements are shared among the tasks. The purpose of system-level synthesis is, thus, to design the system architecture so as to use as few processing elements (including microprocessors and ASICs) as possible without violating the timing constraint for the systems.

There exist already many approaches for system-level synthesis. Several researchers have recognized that importance of combining scheduling, allocation, and mapping in a single algorithm. Wolf and his colleagues were the first to deal with the three problems on operator-level [4] and system-level [5], [6]. They developed an iterative algorithm which first assigns the fastest processing elements to the tasks, and replace the mapping with slower processing elements for the tasks on critical path if the timing constraint is still met.

In this paper, we develop the genetic-based approach for scheduling, allocation, mapping for system-level synthesis. A new chromosome encoding is designed to represent the scheduling, allocation, and mapping solution.

The remainder of the paper is organized as follow. Section 2 defines the terms which will be used in the paper and defines the problem. Section 3 presents the genetic-based approach for the SAM problem. Section 4 evaluates the developed approach and Section 5 concludes the paper.

II. Formal Model and Problem Definition

In SoC, there are more than one processing component in the system. Some of them are designed to execute different software and are called *microprocessor*s; the others are specially designed to complete certain functions and are called *ASIC*s. We call both microprocessors and ASICs processing elements in this paper. Each process element π_j is associated with its normalized cost c_j. We assume that there is no scheduler or sophisticate operating systems to conduct priority-driven scheduling on microprocessors. All the tasks are executed on First-Come-First-Server (FCFS) order.

In this paper, we use the term "task" as it is commonly used in real-time computing community [3]. A *task*, denoted by T, is a sequence of computations, the transmission of a data packet, the retrieval of a file, and so on. A *task flow* denotes a sequence of functionalities to complete a task. For instance, H.264 Encoding in Figure 1 is a task flow and H.264 Decoding is another one. In the task flow for H.264 Encoding, there are six functionalities: PD1, ME, Diff, DCT, Q, and VLC. The operations to complete a functionality are called a *subtask*. A subtask, denoted by τ, is defined by a set of parameters pair, each of which consists of the worst case execution time e and viable processing elements HW. Execution time for a subtask is the amount of time that the subtask needs to complete its execution when it has all the required resources and executes alone. The *viable processing element* for a subtask is a processing element on which the subtask can complete its work. Because one subtask can be executed on microprocessors, different type of ASICs, or DSPs, one subtask may have more than one viable processing element. The following defines the hardware constraint for system-level synthesis.

Definition 2.1 (Hardware Constraint): Each subtask can only be executed on one of its viable processing elements.

The *release time* for subtask $\tau_{i,j}$, denoted by $R(\tau_{i,j})$, is the time instant at which the subtask is known to the microprocessor or processing element, and is ready to execute. We call the maximum allowable response time its *relative deadline*. Except for where it is stated otherwise, by deadline, we mean relative deadline and denote it by d. The following states the deadline constraint for tasks.

Definition 2.2 (Deadline Constraint): Each task cannot start to execute before its release time and the response time of the task is no greater than the relative deadline of the task.

The system-level synthesis consists of three steps: allocation, mapping, and scheduling. *Allocation* for a set of tasks and a set of processing elements determines whether subtask $\tau_{i,j}$ is executed on one of the microprocessors or ASICs. *Mapping*

for a set of tasks and a set of processing elements determines which processing element PE_k is used to execute subtask $\tau_{i,j}$. With a mapping, the number of each type of processing elements used by the tasks are decided. *Mapping cost* for a mapping refers to the sum of the cost for the processing elements which are used in the mapping. *Scheduling* for a set of tasks and a set of processing elements determines the start time and completion time of every subtask $\tau_{i,j}$ on its mapped processing element. A *feasible schedule* means that every task meets its deadline constraint defined by Definition 2.2.

Definition 2.3 (SAM Problem): We are given a set of tasks $\mathbf{T} = \{T_1, T_2, ..., T_N\}$, their task graphs $\mathbf{G} = \{G_1, G_2, ..., G_N\}$, the set of processing elements $\mathbf{\Pi} = \{\pi_1, \pi_2, \ldots, \pi_M\}$. The problem is to determine allocation, mapping and scheduling for the set of tasks so that the mapping cost is minimized subject to the hardware constraint defined in Definition 2.1 and deadline constraint defined in Definition 2.2.

Allocation/assignment and scheduling are each known to be NP-complete for distributed systems [2]. Hence, SAM problem is a NP-complete problem.

III. Genetic Algorithm for SAM

The SAM problem for embedded real-time SoC resembles the scheduling problem for the scheduling problem for distributed end-to-end real-time tasks. The difference is that there is no hardware constraint for distributed end-to-end real-time tasks. We develop a new representation for encoding the solution to avoid generating illegal results which violates the hardware constraint defined in Definition 2.1.

A. Chromosome Encoding

In embedded SoC systems, there are hardware requirement constraints. One subtask may be limited to execute on a set of the processing elements. Using traditional chromosome representation makes it difficult to meet the hardware constraints and may generate many illegal solutions. Hence, two chromosomes are used to encode the solutions. They are *subtask mapping chromosome* and *PE allocation chromosome*.

a) Subtask mapping chromosome: We encode the subtask mapping by ordinal representation. It allows us to apply simply crossover operation to generate legal off-springs. Subtask mapping chromosome is constructed by two lists: subtask reference list and order list. The length of order list and reference list are the number of subtasks in the task set. *Subtask reference list*, denoted by R, is a sequence of integers, each of which represents the index for one subtask. *Subtask order list* for k-th generation, denoted by O_k, is also a sequence of integers, each of which represents the selection of subtask for generating the mapping sequence based on the subtask reference list. Subtask order list for k-th generation is defined as follow.

$$O_k = (o_1 o_2 \cdots o_N) \quad \text{where } 1 \le o_i \le N - i + 1 \text{ for } 1 \le i \le N.$$

Note that every element o_i in the list does not represent the execution order of the subtasks and, only, denotes the order for determining the PE mapping in later steps.

Subtask mapping chromosome for k-th generation, denoted by S_k, is defined as follow.

$$S_k = (s_1 s_2 \cdots s_N) \quad \text{where } 1 \le s_i \le N \text{ for } 1 \le i \le N \text{ and } s_i \ne s_j \text{ for } 1 \le i, j \le N.$$

Given the subtask reference and order list for k-th generation, we iteratively generate the subtask mapping chromosome. For i-th iteration, the o_i-th subtask on reference list is removed from the list and appended into the subtask mapping list.

b) PE Allocation chromosome: PE allocation chromosome for k-th generation, denoted by P_k, is a sequence of integers, each of which represents the position of selecting the processing element. PE allocation chromosome for k-th generation is defined as follow.

$$P_k = (b_1 b_2 \cdots b_M) \text{ where } 0 \le b_j < N$$

Although the above approach avoids generating invalid solutions, the probability for sharing PEs to reduce the cost is low. This is because one PE is shared only when the allocation element b_i is equal to its preceding element b_{i-1}, b_{i-2}, etc. To increase the probability for sharing PEs, while generating PE chromosome, another probability, called *zero insertion probability*, is used. When a generated random number is less than

the probability, the PE allocation element b_i is set as zero. Otherwise, its value ranges from 0 to N. By so doing, one PE may be shared by several subtasks. However, it increases the chance of missing deadline. This is because the chance of being blocked for each subtask increases. Hence, we dynamically adjust the zero insertion probability. When less than one third of the population miss the deadlines, we increase the zero insertion probability. When all the populations miss deadline, we decrease the zero insertion probability. As a result, we may shorten the run-time overhead and increase the schedulability.

c) Allocation and Mapping Sets: Allocation and mapping sets, denoted by AM, represent the allocation and mapping for the subtasks. There are M sets, each of which represents the set of subtasks which are executed on the processing element and has at most N integers.

To construct the sets, we first sort the processing elements in the ascending order of the values of PE allocation chromosome. Suppose that processing element b_k and b_l are the j-th and M-th processing elements in the sorted list. In the j-th iteration for $1 \leq j \leq M$, subtask a_i for $1 \leq i \leq b_k$ and $b_l < i \leq N$ is mapped into processing element b_k if subtask a_i is not mapped into any processing element and processing element b_k is one of the viable processing elements for subtask a_i.

B. Scheduling and Deadline Constraint

The schedule is generated by assuming that all the tasks are released at the same time. Then, the schedule is laid out according to the task graphs. Because we assume that all the subtasks are non-preemptive, the subtasks on every processing element are scheduled in the FCFS order. When more than one subtask is ready at the same time, they are scheduled according to the order in subtask mapping chromosome. After the schedule is laid out, we compute the response time, i.e., makespan, of the schedule to check if the deadline constraint is met.

C. Fitness Function

Each individual is evaluated by a fitness function which shows how good the individual is and the chance to be selected for the next generation. We define two fitness functions: *absolute fitness* and *normalized fitness*. In general, the greater the fitness value is, the better the individual and the higher the chance for being selected. The absolute fitness function for i-th individual, denoted by $AbsFitness(i)$, is calculated based on the cost of the selected processing elements and is defined in the following.

$$\begin{aligned} AbsFitness(i) \quad = \quad & \\ & (\sum_{j=1}^{M} c_j - \sum_{\forall \pi_j \in \{\pi_j | AM_j \neq \emptyset\}} c_j) \\ & - DeadlineMissPenalty \\ & + LastChance. \end{aligned}$$

The first item represents the difference between the sum of cost for all processing elements and that for the selected processing elements. The second item is the deadline miss penalty. The penalty is zero when all the tasks meet their deadlines and is the maximal cost of the processing elements, i.e., $\max\{c_1, c_2, \cdots, c_M\}$, when any task misses its deadline. The penalty is set to eliminate the solution which uses few processing elements and violates deadline constraint. The last item is designed to avoid local optimality. *Normalized fitness* for i-th solution for $1 \leq i \leq PopSize$, denoted f_i, is the ratio of the absolute fitness to the sum of all absolute fitness. In other words,

$$p_i = \frac{AbsFitness(i)}{F} \quad \text{where } F = \sum_{j=1}^{PopSize} AbsFitness(j).$$

D. Population Generation

For each generation, three operators are applied to generate the population from previous generation. First of all, the parents for the new generation are selected from the population in previous generation based on the normalized fitness values. Second, crossover and mutation are applied to the selected parents to generate offsprings.

d) Parents Selection: To generate the population for the new generation, we first select the parents based on the fitness value of each individual. Given the normalized fitness p_i for $1 \leq i \leq PopSize$, the cumulative probability for each individual, denoted by q_i for $1 \leq i \leq PopSize$, is the the cumulative normalized fitness. In other words, $q_i = \sum_{j=1}^{i} p_j$, $0 < q_i \leq 1$ for $1 \leq i \leq PopSiz$, and $q_{PopSize} = 1$. Then, the algorithm generates a random number r in the range $[0, 1]$ for every $PopSize$ times. The i-th solution is selected as one parent when $q_{i-1} < r \leq q_i$ and $q_0 = 0$. The step selects $PopSize$ parents from previous generation. Because a solution having lower cost will have greater fitness value, the chance for

Task Graph Parameters	Values
Task graph in-degree	[1, 2]
Task graph out-degree	[0, 3]
Number of subtasks per task graph	2^n for $1 \leq n \leq 8$
Number of task graphs per design	3
Workload Parameters	**Values**
Task execution time	[1, 20]
Ratio of SW subtasks to HW subtasks	3:7 and 1:1
Number of viable for each ASIC type per subtask	[1, 4]
PE execute time ratio (CPU1, CPU2, ASIC)	(1, 1.5, [0.2, 0.5])
The number of PEs of same type	[1,3]
The number of PEs (including CPU and ASIC)	5, 15, 30
PE cost ratio (CPU1, CPU2, ASIC)	(1, 0.2, [0.01, 0.06])
Genetic Algorithm Parameters	**Value**
Population Size	30
Crossover selection Probability	0.35
Mutation selection Probability	0.15
Termination Threshold	0.05%

TABLE I. Task Graph Parameters and Workload Parameters

the solution to be selected as the parents will be greater.

e) Crossover and Mutation: Two genetic operators, crossover and mutation, have been used to evolve new generations. The *crossover* operator creates two new chromosomes by crossing over two parent chromosomes by selecting random cross point. The crossover is applied to subtask order list and PE allocation list. Note that because of the well-designed subtask order list, no special care is required for off-spring generation. The *mutation* operator simply modifies a chromosome by altering the data bits in of the selected chromosome. The probability for selecting one individual for crossing over and mutation are *crossover selection probability* and *mutation selection probability*, respectively. The two probabilities are the parameters for genetic algorithms.

E. Termination Conditions

The genetic algorithm terminates when the difference between the maximum absolute fitness value of previous generation and current generation is less than *termination threshold*. The condition is applicable when the deadline constraint is met.

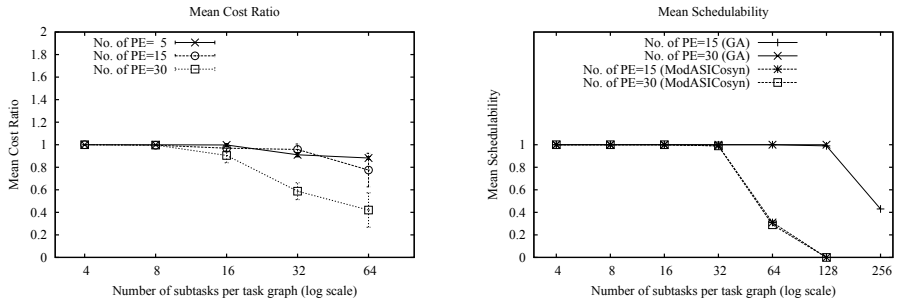
IV. Performance Evaluation

We evaluate the genetic-based approach with extensive simulations and compare the results with the algorithm developed by Wolf etc. [4], [6]. The algorithm [4] developed by Wolf etc is designed to solve the SAM problem on operator level and Algorithm ASICosyn[6] is designed for single task graph with custom ASICs. We modified the algorithm so as to solve the SAM program for system-level synthesis with multiple task graphs. However, the concepts of solving the SAM program remains unchanged. For sake of simplicity, we call the modified algorithm $ModASICosyn$ algorithm.

f) Workload Parameters: The task graph is randomly generated by TGFF [1]. The workload parameters and the parameters for generating task graphs are summarized in Table I.

Three metrics are measured to evaluate the developed algorithm: mean cost ratio, mean schedulability ratio, and run-time overhead. *Mean cost ratio* is the ratio of the cost of the solution by genetic algorithm to that by $ModASICosyn$ algorithm. When the mean cost ratio is greater than 1, the solution by the developed algorithm has greater cost than that by $ModASICosyn$ algorithm. *Mean schedulability ratio* is the ratio of the number of schedulable solutions by the algorithm to the number of generated samples. *Mean run-time overhead* measured the times used by the algorithm to find the solution. The simulations are conducted on personal computers which have Intel Pentium IV 3.0 GHz processors, 512MB RAM, Fedora Core 4 installed.

g) Evaluation Results: Figure 2 shows the performance evaluation results. Figure 2(a) and (b) shows the mean cost ratio and mean schedulability for the case that the number of software subtasks and hardware subtasks is 1:1. The range in Figure 2(a) shows the interval for 95% confidence interval. In Figure 2(a), only the results for 4 to 64 subtasks per task graph are shown. This is because



(a) Mean Cost Ratio (SW:HW=1:1) (b) Mean Schedulability (SW:HW=1:1)

Fig. 2. Evaluation Results

the $ModASICosyn$ algorithm fails to find feasible results for larger input instances. The results confirm that the genetic algorithm are suitable for the case that the solution space exponentially grows and the size of input parameters are greater. In Figure 2(b), we can see that the genetic algorithm continue to find almost all the solutions. However, $ModASICosyn$ algorithm fails to find all feasible solutions when the number of subtasks is greater than 64. The results for the case that the ratio of software subtasks to hardware subtasks is 3:7 are similar and, hence, are not shown.

Figure 3 shows the run-time overhead for the genetic-based algorithm. The results show that

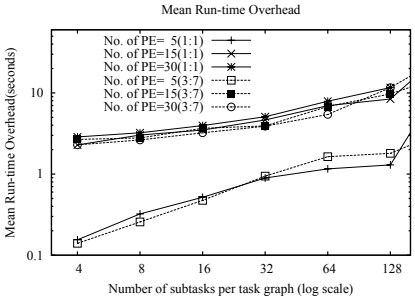


Fig. 3. Run-time Overhead

the number of PEs impose negligible effects on run-time overhead. However, the number of subtasks causes dramatically increase on run-time overhead. This is because when the number of subtask doubles, the total number of subtasks in the problem is sextuple. We consider that it is acceptable to find the feasible solution for 354 subtasks on 15 PEs in one minute.

V. Summary

We develop a genetic-based approach for system-level synthesis for scheduling, allocation, and mapping. The approach determines the scheduling, allocation, and mapping for the system at every step to find the near optimal solution. In addition, a new chromosome encoding is designed to avoid generating illegal system architecture and, hence, reduce the number generations to obtain near optimal solutions. We evaluate our approach with extensive simulations and compare our performance with other approaches in the literature. The results show that the genetic-based algorithm outperforms other algorithms when there are large number of instances and its run-time overhead is still acceptable.

References

[1] R. Dick, D. Rhondes, and W.Wolf, "TGFF: Task graphs for free," in *Workshop Hardware/Software Codesign*, 1998.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.

[3] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[4] Richard J. Cloutier and Donald E. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," in *Proceedings of the 27th ACM/IEEE conference on Design automation table of contents*, Orlando, Florida, United States, 1991, pp. 71 – 76.

[5] W. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Transactions on VLSI Systems*, vol. 5, no. 2, June 1995.

[6] Yuan Xie and W. Wolf, "Co-synthesis with custom asics," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2000, pp. 129 – 133.

Reducing System Entropy Using Fine-Grained Reboot

Hiroo Ishikawa Harry Sun Tatsuo Nakajima
Department of Computer Science
Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan 169-8555
{ishikawa, sunlei, tatsuo}@dcl.info.waseda.ac.jp

Abstract

An embedded system which is a part of a whole product should guarantee its reliability. Recovery support at an operating system level is necessary to reduce the cost of testing because software systems have been becoming so large and complex that software testing is imperfect. This paper proposes an operating system with fine-grained rebooting as its recovery support, called ArcOS. ArcOS is built on a microkernel architecture, and each process is made of modules, called cells that allow a part of a process to reboot. While a simple reboot discards all the state of a cell and thus leads system inconsistency, we discuss policies and mechanisms to reboot cells safely.

1. Introduction

Software for embedded systems is required to satisfy the reliability as a part of a whole product. Since software doesn't have solid state properties and its state is very complex, a failure caused by race conditions, resource leaks, Heisenbugs, and environment-dependent bugs occurs in a random manner in nature. Thus it is dif cult to estimate the reliability in a software system. We term the randomness the *system entropy*. If there is a system that system entropy is low enough, its reliability is high. In order to reduce the system entropy, operating system support is necessary because runtime behavior mainly affects the system entropy.

Rebooting is referred to restarting a system with the initial state, and is used as a reliable method to cure the system inconsistency in some systems[3][6][12]. In addition, rebooting is often used to recover a system from crash or freeze[5][14]. Thus, Rebooting is an appropriate technique for recover the transient and unpredictable failures, hence reduces the system entropy. However, only several OS research projects have studied on rebooting mechanisms, policies and applications. We need to address these issues and provide rebooting as a method for reliable operating systems.

Our basic idea is the ne-grained rebooting which allows a process such as a server or an application on microkernel to partially restart with the initial state. Microkernel and virtualization architecture have been recently popular as reliable software infrastrctures because they isolate OS services or multiple OSs from each other[15]. Even if a failure occurs, only the server or the OS that suffers from it crashes and reboots without propagating errors to other servers or OSs unlike monolithic kernel OSs. However, it is unsolved that services are unavailable during a reboot. For example, a virtual machine architecture requires an entire OS reboot. A microkernel architecture is better than the virtual machine architecture because it can reboot a process individually. However, rebooting a process is rather longer if it is used for consumer electronics. The ne-grained rebooting deals with the runtime errors without completely restarting a process. We think that it improves the availability of embedded systems.

This paper proposes ArcOS, a self-rebootable operating system based on the microkernel architecture. With the rebootable property, ArcOS cannot only recover from the transient errors, but also cure its anomalous behavior such as memory leaks and performance degradation. Although a simple module rebooting may corrupt a process, ArcOS provides basic mechanisms to avoid the inconsistency by using *continuations*[9] for threads and a tree-based memory management. They reduce the tasks at a reboot by maintaining the state of cells at runtime. These mechanisms are used both in a proactive way (*i.e.* software rejuvenation), and in a reactive way (*i.e.* failure recovery).

The paper is organized as follows. The next section describes related work in terms of utilization of rebooting especially inside operating systems. Section 3 describes the overview of ArcOS. In Section 4, we discuss the cell component model. Section 5 describe the design of ArcOS including the rebooting mechanisms. Section 6 shows the current status of our development. We developed a le server for the case study. In Section 7, we discuss the further is-

sues. Section 8 concludes the paper.

2. Related Work

Rebooting is studied as a method to improve reliability or availability of a system. A proactive reboot can discard the inconsistency due to a ws in a system and prevent it from failure[6][11]. Eventually, proactive rebooting makes downtime shorter than failure recovery does.

Rebooting is not only in a proactive manner but also in a reactive manner, and some kinds of OS support have proposed so far. At an early stage, the OS support was studied on UNIX, such as Recovery Box[2] and Rio File Cache[7]. These systems basically react to OS crash failures, and quickly recover the system keeping the contents of RAM over a reboot. Since these systems are made for UNIX, all the OS services need to be stopped when a failure occurs.

It is possible for microkernel OS to recovery or prevent failures without stopping entire OS services[15]. For example, ChorusOS provides persistent memory and ne-grained checkpointing mechanism as OS rebooting support[1].

The a ws in device drivers are the major cause of the failure of Linux[8]. Nooks provides a solution to this problem by isolating device driver from the Linux kernel[14]. Nooks uses rebooting drivers to recovery from the driver crash. In addition, it provides shadow-drivers to prevent application crash from rebooting drivers[13].

3. ArcOS Overview

ArcOS is a multi-server operating system based on L4 microkernel. Its architecture is shown in Figure 1. It provides the process, memory, le, monitoring, and rebooting servers, and device drivers. Every application, server and device driver resides in a separate address space, and is constructed of one or more cells that are dynamically loaded from disks or RAM, and bound at runtime. Basic abstractions such as threads, IPCs, and address spaces depend on L4. This paper doesn't address the detail of the servers and device drivers apart from the rebooting server.

4. Cell: A Unit of Reboot

A *cell* is a unit of reboot. Cells are dynamically loaded into process's address spaces and form a process. A cell or a group of cells is rebooted at an exceptional state during its execution or when anomalous behavior is observed by a monitoring service. The concept of cells is illustrated in Figure 2. A cell has a set of code and static data sections. A global memory area is provided to cells. A cell can dynamically allocate memory by importing a memory blocks

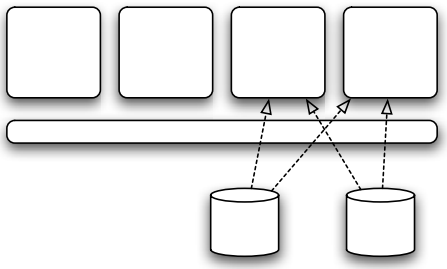


Figure 1. Overview of ArcOS

into its protection domain. Cells communicate via function calls with the continuations[9]. A thread can run through multiple cells.

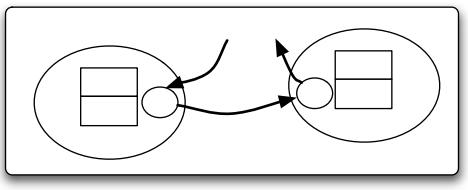


Figure 2. Cell Component Model

4.1. Requirements

There are three requirements for cells and the rebooting service.

Keeping system state consistency over a reboot: This requirement necessitates the state management, which provides how to and what kinds of state to save over a reboot, and the dependency management, which recognizes which cells to reboot at the same time.

The simplest module reboot reloads and restarts it with the initial state. This obviously discards all the state including unanticipated state. However, as long as it holds a state over time, it breaks the consistency with other cells and thus may break an entire system consistency.

Keeping the consistency doesn't always mean saving a cell's state. If a cell saves its state before its reboot, it can restart in the middle of its execution with the saved state. If a cell doesn't save its state, it is possible to keep the consistency by rebooting dependent cells at the same time. This is useful when a group of cells are well-separated from other cells, and it is likely to happen.

Rebooting as quickly as possible: The state and dependency managements have to be simple to shorten the pro-

cess of rebooting. Although saving state and resolving dependency are necessary, they sometime take a long time and turn out to be a longer down time. Simplicity of the rebooting process has to be taken into account.

Being relocatable: This is an optional requirement but is useful if RAM is physically damaged, or if a series of illegal memory accesses happens continuously. A relocatable cell can be reloaded into a different address during its rebooting process.

4.2. Memory Management

A cell can obtain a dynamic memory region from the address space where it resides. The obtained region is released when the cell explicitly releases it or is rebooted. The allocation information is described in a cell so that discarding the information releases the corresponding memory region. Thus, the memory management at a reboot becomes much easier.

The region once allocated is inaccessible from other cells; it cannot be shared among more than one cell. Cells can share data by passing a region that a cell owns to another cell. This method reduces the dependency between cells, thus makes a cell reboot easier.

4.3. Execution

Threads are created inside cells. Each process has a root cell where the initial thread starts. The other threads are created as children of the initial thread.

A thread saves its cell-local stack at an inter-cell call. The saved thread state is termed a *continuation*. With continuations, an inter-cell call never returns; the program code of a caller is divided into a pre- and a post-call functions. The continuation is restored when the thread re-enters the cell.

4.4. Exception Handling

Cell rebooting itself works transparently from applications, however, it has to be determined in accordance with an application's policy. There are three types of exception state. The first type is the exception that a user program can deals with and recover by itself, such as a user's invalid input. The second type is the exception that is expensive to deal with because the exception handling includes many routines or needs to wait for a long time. The last type is the exception that no methods except rebooting can handle such as hang or crash. There is an additional exception that is thrown as a result of long-term system monitoring such as memory leaks. While the cell rebooting can be transparently performed in the third and forth cases, it should follow an application's decision in the first and second cases.

5. Design

5.1. State Management

We define three states and two characteristics of a cell that must be taken into account at a reboot. The combinations of these states help to identify if rebooting is reasonable.

Thread State: At the moment of a reboot, a cell is in either the state where no threads are running, or the state where more than one threads are running. If threads are running on the rebooting cell, it is necessary to wait for them going out of the cell, or to force them to stop and restart from the previous cells after the reboot is finished. If a thread runs only inside a cell, it is simply terminated and its stack is discarded.

Allocation State: A cell can dynamically allocate memory on the heap for itself. The memory is discarded at a reboot, or kept over a reboot if it is sent to another cell.

Sharing State: A cell can share resources. Cells depend on each other through shared resources. Shared resource involves data on the heap, I/O and locks. A reboot can corrupt shared resources if it is performed during manipulating the data.

Mobility: A cell can be rebooted in a different area of the address space. In that case, the dependent cells also must be rebooted to relocate their references to the rebooting cell. Shared resources and the continuations also must be maintained.

The simplest rebooting is to reload and overwrite the cell that is stateless and under the condition of none of threads running inside it. In this case, it is unnecessary to reconfigure all the symbols in the rebooting cell and the references in other cells. For example, a cell that only prints out messages to a console, or a cell that translates information (e.g. inode number to block number) are rebooted in this way.

The most difficult rebooting is to reboot a cell that has all kinds of state such that multiple threads are created, multiple thread including external threads are running, some regions of memory are allocated, some resources are shared with other cells, and a move after rebooting is required.

5.2. Types of Rebooting

We define three types of rebooting. Each of the type is performed under different condition, and its effect to the system is different.

Single Reboot This policy attempts to reboot a single cell even if there is dependency. To keep the consistency between cells, this policy maintains the state of the rebooting cell over a reboot. Therefore, the cell should be as simple as possible. A complex cell may not be able to be handled correctly.

Multiple Reboot The multiple reboot selects the dependent cells as keeping the consistency of a process. The dependent cells are determined with call graph and data o w graph.

Whole Reboot The whole reboot policy reboots an entire process. All the state of the cells in a process are discarded.

5.3. Thread Management

Continuations are used for communications between cells. A continuation is originally referred to the *rest of the computation*. Using the continuations, no function is ever allowed to return. We use the continuations as check points to restore computation after reboots. Eventually, the continuations reduce the dependency between cells, and every cell can be rebooted at any time. When a function in the rst cell invokes a function in the second cell, a continuation is used. A normal function call in C language assumes that the call will return to the calling address. However, a call from a cell to another cell doesn't assume that it will return after the method completes its procedure. This *never-return* concept is similar to the concept of the crash-only software proposed by Candea et al[4].

At each inter-cell communication, the context of the invocation is stored in the special storage in the caller cell, and the identi er of the caller cell is stored in the thread control block. When a reboot is performed running in the callee cell, the thread context is restored using the identi er and the contents of the storage, and then the thread is restarted with the context after the reboot is nished.

5.4. Memory Management

The goal of the memory management is to let a cell have its own memory management information. This allows to release the memory a cell owns when it is rebooted.

The memory management system is shown in Figure 3. The memory management cell takes in charge of the total memory management. But the memory management information is distributed to the other cells. The memory management cell holds references to the other cells. Each cell manages the references to the heap as a tree. Each node include the address, the size and the permission of the region. all nodes are linked together in address order to facilitate

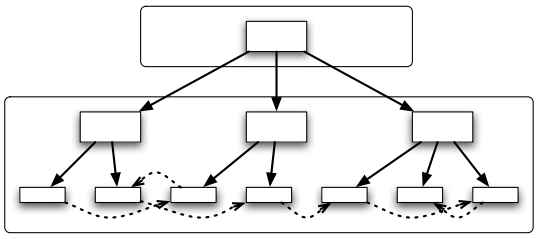


Figure 3. Memory Mangement Tree

memory allocation. The memory allocation policy is rst- t. To share data between cells, a node is transferred from one to another. When a node is transferred, it is removed from the sender's tree and then added to the receiver's tree.

When a reboot is performed on a cell, the memory management information is simply discarded. Since this corrupts the linked list, the memory management cell repairs the list by walking through the memory management trees.

6. Implementation Status

We have been developing ArcOS services on top of L4 microkernel. The design of ArcOS depends on the functionalities provided by L4, such as threads, IPC, address spaces, and memory mapping. While each cell communicates using the function call with the continuations, the communications over the process boundary is performed by using IPC provided by the microkernel.

The rebooting service that we have developed runs as a group of processes on the microkernel, and is in charge of loading and reloading cells. A cell is loaded from a disk or RAM. When an exception is caught or a process commands to reboot, the rebooting service reboots the cell.

7. Further Discussions

Although our basic ideas are shown above, several issues in order to use the rebooting mechanisms still remain.

7.1. Protection Domains

Our memory management doesn't protect illegal accesses. Cells in the same process can read and write anywhere in the heap no matter which cell owns. This is because current MMU deals with 4k-byte boundary at minimum. In order to protect memory nodes from the illegal accesses, we need to think about software-based approaches, for example, proposed by [10] or [16].

7.2. Dependency Analysis

When a group of cells needs to be rebooted, the system must analyze the dependency between cells. Currently we only use call trees and the loading order for the analysis. Cells share many kinds of state with each other, and this cause a problem when rebooting. For example, assuming that two cells share a memory node, then the rst cell passes the node to the second cell. Since the memory node cannot be shared at the same time. The node and data are discarded if the second cell is rebooted at the moment. In this case, it is necessary to analyze the data o w and save the node prior to the reboot. Dependency analysis need to consider not only data o ws, but also locks, I/O, and so forth.

7.3. System Monitoring and Behavior Analysis

Rebooting can recover performance degradation. System monitoring and analysis are necessary to recognize this type of errors and determine whether to reboot a cell. Using this functionality, an OS can deal with errors in a proactive manner. However, current methods for system monitoring require a large number of data and statistics in advance to running a system. We need to think how to analyze a system's behavior based on a little knowledge. This is necessary for dynamically evolving systems.

7.4. Preventive Reliability vs. Corrective Reliability

Rebooting is usually used as a corrective method. However, as the research on software rejuvenation has been done, it can be used as a preventive method. Each of them has several advantages and disadvantages. For example, in a corrective method, a system can easily determine when to perform a cell reboot. On the other hand, it has to recover the failure. In a preventive method, a system can predict when to reboot based on either system monitoring or periodic rebooting, and avoid failures. However, the preventive method brings the problem of electricity consumption that CPU and disks utilizations stay high. It may be hardly acceptable for battery-powered embedded systems.

7.5. Development Support

ArcOS expects inter-cell invocations to use the continuations for keeping system consistency over a reboot. However, this makes programming more complicated. In C programming language, the continuations can be written using `goto` command or function pointers. While these methods allow the OS to reboot a cell at many points of execution path, the visibility of program code becomes worse. To solve this problem, we may need to think about extensions of C language or other programming languages such as Scheme or Ruby, which supports the continuations as rst-class abstraction.

8. Concluding Remarks

This paper has described an operating system support for ne-grained rebooting. Embedded software needs to be highly reliable, however its system entropy increases because of its complexity. Runtime support is necessary to reduce the system entropy. Rebooting is an effective method to recover from and prevent transient errors that increases the system entropy. Therefore, it is considered one of the techniques to deal with the system entropy.

ArcOS is a reliable operating system that supports rebooting as its rst-class functionality. A process of ArcOS is constructed of rebooting units called cells. We have described that the problems of state management can be solved with the continuations, memory sharing, and policies of rebooting. We are currently developing the rebooting services and are developing ArcOS services that run on L4 microkernel.

References

[1] V. Abrossimov, F. Herrmann, J. C. Hugly, F. Ruget, E. Pouyoul, and M. Tombroff. Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology. Technical report, Chorus Systems, Inc., 1996.

[2] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings of Summer 1992 USENIX Conference*, June 1992.

[3] T. Boyd and P. Dasgupta. Preemtive Module Replacement Using the virtualizing Operating Systems. In *Proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems*, June 2002.

[4] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[6] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive Management of Software Aging. *IBM Journal of Research and Development*, 45(2), March 2001.

[7] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM symposium on Operating systems principles*, October 2001.

[9] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. Technical Report CMU-CS-91-115R, School of Computer Science, Carnegie Mellon University, October 1991.

[10] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi Single-Address-Space Operating System. *Software: Practice & Experience*, 28(9):901–928, July 1998.

[11] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of The 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[12] K. Lee and L. Sha. Process Resurrection: A Fast Recovery Mechanism for Real-Time Embedded Systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RSTAS'05)*, March 2005.

[13] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, December 2004.

[14] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[15] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51, May 2006.

[16] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Ef cient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, December 1993.

Static Analysis Support for Measurement-based WCET Analysis

Stefan Schaefer [◇†] Bernhard Scholz [†] Stefan M. Petters [◇‡] Gernot Heiser [◇‡]

[◇] National ICT Australia*

[†] School of IT
University of Sydney

[‡] School of Computer Science
and Engineering
University of NSW

{*stefan.schaefer*,*stefan.petters*,*gernot*}@nicta.com.au
scholz@it.usyd.edu.au

Abstract

Guaranteeing that the worst-case scenario has been covered for each basic block individually is a major challenge in measurement-based WCET analysis. On the static analysis side the accuracy of hardware models used are also subject to doubt as they are based on available documentation provided by vendors which may be not detailed enough nor correct or both. Even for verified models the question remains whether subsequent chips adhere to the same specification. We introduce a new approach to enhance measurement-based WCET analysis by deploying static analysis to ensure test coverage on basic block level and reduce pessimism. In particular, our work focuses on the questions how detailed does the hardware specification have to be to make the measurements trustworthy.

1 Introduction

Response time analysis for real-time systems requires the *worst-case execution time* (WCET) of all code in the system to be known *a priori*, but finding the WCET of programs is difficult at best. Current microprocessor architectures are highly complex as they incorporate pipelines, branch prediction units, multiple execution units and caches.

There are currently two major approaches for WCET analysis: (1) static analysis of programs and (2) measurement-based techniques. Static analysis involves modelling of the target architecture. Current CPU architectures feature a rich set of mechanisms to accelerate the execution time of programs, e.g., caches, pipelines and branch prediction units. Most of these features are transparent for programmers and programs and act in a speculative way. Models of processors are pressed hard to keep track of all these features, but due to a complexity problem most models do not implement them and place extremely heavy restrictions on code, which results in very pessimistic upper WCET bounds and compensates the advantage that it does not depend on actual execution-time measurements of code.

Creating an accurate and precise model for static analysis is difficult at least, as:

- Vendors lose flexibility once they published specifications. Unpublished specifications can be subject of a future change much more easier.
- Handware manufacturers keep their economic benefit in mind. In most cases, manufacturers do not benefit from publishing specifications.
- Finally, the translation from documentation to the model used for static analysis is also an error prone process.

This has to be seen against the backdrop of an increasing number of real-time systems, in particular those deploying high performance architectures invading every day life.

Measurement-based analysis techniques are still current practice in industry. In principle, execution time measurements are taken with an assumed worst-case input. Besides being tedious, this does not give a satisfying answer for the WCET because it is impossible to enumerate all possible (infinite many) inputs for a pro-

*National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

gram and measure each execution time. Measurement-based analysis techniques always raise the question whether there exists a more pessimistic scenario, in which the execution time exceeds the WCET observed so far.

The major problem with that approach is that the execution time of a basic block differs when different input parameters are provided as different input causes different paths to be taken to reach this basic block during execution *and* the correlation between input and execution time is not obvious. The worst-case behaviour of a basic block can be exposed easily, but this is generally considered as not being suf cient since it is hard to prove that this is the case.

Our approach aims to combine the strength of the static and measurement-based approaches and involves two steps: In a rst step, measurements of basic block execution time pro les (ETPs) are taken through measurements. Static analysis methods will validate these ETPs with the help of a simple hardware model. If the validation fails, the static analysis requests more measurements with different input. In a second step, ETPs of larger syntactical constructs are computed. Finally, static analysis comes in to improve the tightness of the upper WCET bound.

We develop and test our approach within NICTA's Potoroo project, which aims to establish the WCET of all code of the L4 microkernel. Performance and trustworthiness are the main objectives of the L4 kernel implementation used within the project. Trustworthiness covers functional as well as temporal behaviour. Being able to estimate the WCET of these systems' microkernels makes them trustworthy in the temporal sense, while minimality is an important step to achieve functional trustworthiness, as a formal proof of functional correctness, as anticipated by the project L4.veri ed, is only possible if the code base is small.

Our report is organised as follows. In Section 2 we survey the related work. In Section 3 we introduce our new hybrid approach and in Section 4 we draw our conclusion.

2 Related Work

Pure static WCET analysis methods try to nd the greatest execution time explicitly by computing of implicitly by n d the execution path that causes this execution time. It is well accepted that nd this longest path is undecidable in the general case. This does not hold for real-time contexts due to heavy restrictions on the source code like bounded loops, absence of dynamic structures and forcing of annotation were made such as in the works of Chapman et al. [5] and Blieberger et al. [4]. Engblom et al. introduce *Co-Transformation* [7] to establish execution time informations between the source code and the object code in order to overcome the massive restructuring of code done by the optimisation stages of compilers.

Timing schemata were used by Lim et al. [11] to model the WCET behaviour of RISC processors. Our approach uses timing schemata as well. The general idea of a timing schema is to associate to each program construct a worst-case timing abstraction that accounts hardware features that affect the WCET.

Formal methods such as *Symbolic Analysis* [1], *Code Transformation* [12], *Model Checking* [15] and regular expressions [13] are other approaches to obtain an upper WCET bound through static analysis.

The most important technique is *Integer Linear Programming* (ILP). The problem of nding the longest execution path into an ILP problem is done by Li and Malik [10], which avoids the explicit enumeration of all feasible paths of a given control ow graph (CFG). Their implementation `cinderella` performs static WCET analysis by that approach and targets the Intel i960 processor.

Another ILP-based approach, `aiT`, is discussed by Ferdinand [8]. This tool computes an upper WCET bound based on a detailed hardware model. The main feature is its use of abstract interpretation. However, this tool puts some restrictions on the programmer, i.e., the absence of dynamic structures. Furthermore, it does not consider task switches, threads, parallelism and speci ed return addresses. The major drawback of ILP-based approaches in general is that all ILP solvers can only give an upper WCET bound, but cannot conclude to the actual path that caused this speci c execution time.

Hardware simulation is a technique used by Colin and Puaut [6]. The authors introduce a framework for WCET analysis based on CFGs and *simulate* instruction caches, branch prediction units and pipelines statically to obtain the WCET of a program.

White et al. introduce a framework to bound worst-case instruction cache performances and a tool that bounds the worst-case data cache performance predictions [16]. While timing predictions for instruction caches with pipeline simulation are as tight as predictions for direct-mapped caches, the worst-case data

cache performance prediction can be tightened signi - cantly.

Another tool for estimating WCET with a hybrid method is pWCET [3]. This tool performs probabilistic worst-case execution time analysis. It combines static analysis and measurement-based methods and consists of two parts. The basic block level ETPs obtained through measurements are combined with the help of the computational tree and timing schemata in order to obtain an overall ETP. The approaches developed within the Potoroo project and the approach described in this report are extensions of the approach in pWCET, in particular regarding obtaining WCETs for compound syntactical constructs, and the reader is referred to the publications above for details on the measurement-based approach.

The most relevant related work is [9]. Kirschner et al. introduce a new hybrid method that combines static analysis and measurement approaches. This technique guarantees that every feasible path of the target program is executed and its execution time is measured. The static analysis performs o w fact extraction from the source code and modelling of an execution-time model based on the target architecture. An abstract syntax tree (AST) traversal annotates sub-trees with upper WCET bounds. A WCET bound for a compound construct is computed from the WCET bounds of all sub-constructs with the help of timing formulas. Due to the nature of an AST, only local ow informations can be used. A path-based search focuses on a bottom-up WCET estimation. The third method, *Implicit Path Enumeration Technique* (IPET), transforms the structure of a program into a set of constraints of the CFG. ILP is then used as the back-end to solve the WCET problem. With the help of this technique, all possible execution paths are considered implicitely. However, this is essentially an exhaustive search which becomes quickly infeasible even for code with very moderate complexity. Furthermore, execution time is not just dependent on control o w, but is also substantially in u enced by data locality (i.e. caching effects), which is not considered in that work.

The thesis [14] involves an exhaustive search combined with measurements. In this case, control ow during measurements was enforced. To deal with complexity, partintiong the application into measurement blocks provided a manageable number of paths to be explored. These blocks were insulated against each other by forcing a worst-case state of caches and branch prediction between measurement blocks, making sure the worst case has been captured.

3 Approach

3.1 Measurement-Based Analysis

The measurement-based approach this work complements relies on execution traces gathered during execution of the code. The traces may be generated using instrumentation code, debug tracing tools, or cycle accurate simulators. However, cycle accurate simulators suffer the same uncertainty of their models as pure static analysis and is hence not considered a good solution. The traces contain tuples of a time stamp and an basic block identi er , which connects the time stamp with the execution of a basic block. The basic blocks are matched to their corresponding nodes in the computational tree that is obtained from the CFG of the executable. In this manner, we get an execution-time pro le (ETP) for each basic block. Such an ETP reveals cache misses occured during execution through their execution time penalty. Basic block ETPs are then combined to ETPs of higher order nodes in the computational tree with the help of timing schemata and surpremal convolutions. Surpremal convolutions are mathematical operators that describe the convolution of the ETPs such that the resulting ETP considers every dependency between the two argument ETPs. In this manner, we cover any possible dependency between the two basic blocks. The root node of the tree contains the ETP describing the WCET behaviour of the entire code. The details of the whole analysis method can be taken from [2]. Within this report, we will focus on the ETPs of basic blocks and issues when combining them. Picture 1 depicts an overview of the operations performed during WCET analysis.

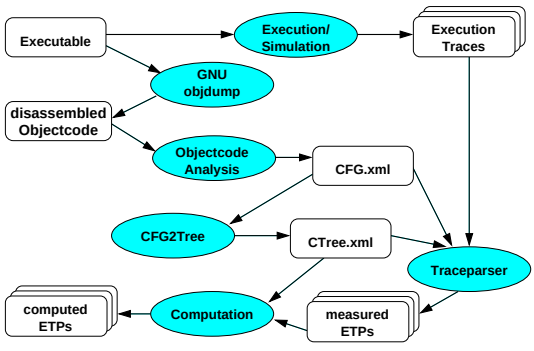


Figure 1: Our Current Tool Chain

3.2 Coverage Checking

As mentioned before, the major problem of approaches based on basic block level measurements is to guarantee that suf cient measurements have been taken. Our proposed approach is to supplement the measurement-based WCET analysis by static analysis. The static analysis part is focusing on rst order effects like caches. Second order effects like pipelining are not considered. The main motivation of this limitation is that any risk involved in underestimating the WCET, if all cache effects have been surely covered, is very small. The variabilities of rst order effects is large in relation to the those of second order effects.

In a rst stage our aim is to predict possible numbers of cache misses in basic blocks. Creating a model of caching behaviour is easy compared to, for example, the interlocking dependencies between pipeline, cache and arithmetic model. It can be ported easily and the complexity pressed upon the computational stage is kept low. Cache size, associativity, and cache line size can usually be taken straight from the data sheets, but the replacement algorithm if often not speci ed or given as "pseudo-random". This allows the vendors for some leeway in changing implementations without changing documentation. In the end, only a small number of replacement algorithms are used, with the *clock algorithm* and *least recently used* (LRU) being very popular. All these can easily be identi ed and veri ed by simple test programs testing for the boundary cases.

The static analysis stage decomposes the object code of executables into basic blocks and keep track of memory accessing operations within each basic block. In this manner, a cache miss pro le for each basic block is obtained. The results of the cache prediction model are then compared to the ETPs observed during measurements. The granularity difference between caching and other effects allows an easy comparison between the ETPs and the model results.

Our hybrid approach uses the idea of a feedback mechanism as shown in Figure 2. As indicated, more measurements are taken if the measured outcome deviates from predictions. If a basic block is not exhibiting the predicted worst-case behaviour for an extended period of time the process is stopped and manual intervention necessary. The manual intervention can be either to craft input data to create the worst-case scenario for an basic block or to dismiss such a worst-case scenario as being impossible to achieve. The latter may happen as the approach so far does not take infeasible paths into account in the computational stage. However, we aim at minimising such interventions to make the problem managable.

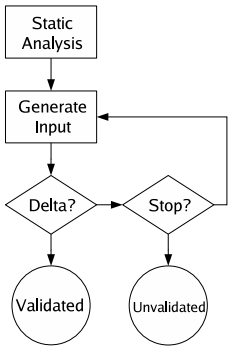


Figure 2: The Feedback Loop for our Hybrid Approach

Figure 3 shows the correlation of the cache misses of one basic block. The upper part shows the execution time penalties predicted by the static analysis. These penalities are variable as the cache itself has a variable execution time when writing back a cache line. The lower part shows the clusters of execution times around cache misses observed during running the basic block. These clusters arise from second order effects like pipelines. In order to etablish these correlations, matching the cache miss pro les and the ETPs is the key point. As indicated, the predictions obtained from the hardware model may differ from those obtained from the ETP in which case more measurements are necessary. The granularity that separates one cluster from a neighboring cluster is one of the most fundamental issues on this matching problem.

As pointed out, our aims are microkernels running on embedded systems. In particular, the L4 microkernel developed at NICTA has system call functions whose execution times in general are short since not much code is involved. This assists to minimise the space of possible cache states that might be achieved during execution. As a matter of fact, we have very little to no eviction of cache contents.

3.3 Overestimation Reduction

In order to obtain save results, the measurement-based approach makes no assumptions regarding dependency of execution times between basic blocks, but rather makes sure that any form of dependency is conservatively covered through surpremal convolutions. This

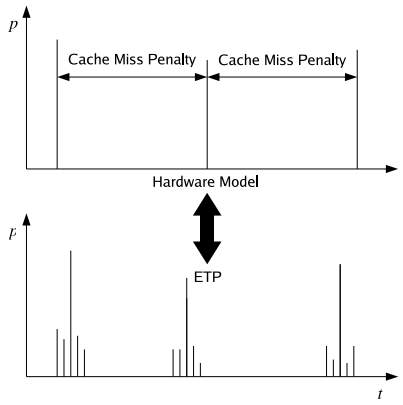


Figure 3: Correlation between Model and Execution Time Pro le

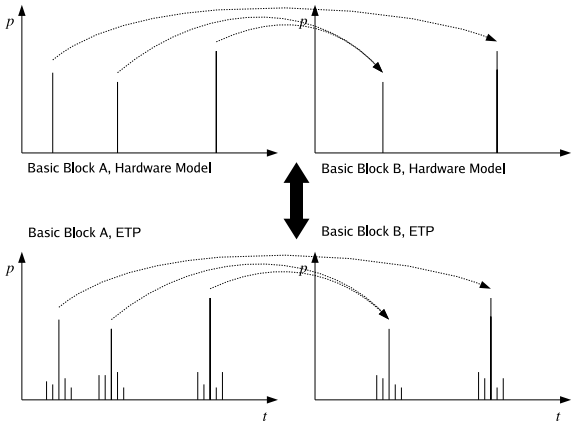


Figure 4: Establishing Dependency Structure

leads to inherent overestimation.

As a next step after verifying that suf cient measurement data has been obtained, our approach focuses on identifying dependencies between execution times of basic blocks. In the previous section we have described how to establish a connection between an ETP and a cache miss pro le. In this step we establish now dependencies between the cache miss pro les of different basic blocks, which can than be taken into account in the computational stage of the approach using the measured ETPs.

Figure 4 depicts the established cache miss dependencies between two basic blocks A and B. The upper part shows the dependency establishment with the help of the hardware model. These dependencies may then be considered when combining the ETPs of the two basic blocks using timing schemata, which is shown in the lower part. The space of possible cache state transitions is reduced massively such that we can make statements like "If this cache miss occures in A, then it will not occure in B".

4 Conclusions

In this report we have detailed our work to enhance trustworthiness of measurement-based worst-case execution-time analysis by deploying static analysis techniques. In order to avoid a common problem of lack of trustworthiness of the models deployed in static analysis, we aim to use only a minimalistic model considering caching, opposed to modelling the entire processor and peripheral units. This model can be easily validated on any given hardware platform. Building on top of the trustworthiness analysis of the measured ETPs of basic blocks, we also aim to use the technique to tighten up on existing sources of overestimations by describing dependencies between basic blocks which can then be used in the later computational stages. Future work will focus on implementing and validating this approach.

References

[1] Guillem Bernat and Alan Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming. Palma (Spain).*, May 2000.

[2] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, December 3-5 2002.

[3] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, April 2003.

[4] Johann Blieberger. *Timing Analysis of Object-Oriented Real-Time Programs.* (submitted), Institute for Informatics, Technische Universität Wien, Vienna, Austria, 1995.

[5] Roderick Chapman, Andy Wellings, and Alan Burns. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada, June 1994.

[6] Antoine Colin and Isabelle Puaut. A Modular and Retargetable Framework for Tree-based WCET Analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, Netherlands, June 13-15 2001.

[7] Jakob Engblom, Peter Altenbernd, and Andreas Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *The 10th Euromicro Workshop on Real-Time Systems (ECRTS98)*, Berlin, Germany, June 1998.

[8] Christian Ferdinand. Worst-Case Execution Time Prediction by Static Program Analysis. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, USA, April 26-30 2004. IEEE Computer Society.

[9] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. UBooks Verlag, Dec. 2005.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461. ACM, June 1995.

[11] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Trans. Software Eng.*, 21(7):593–604, 1995.

[12] Hemendra Singh Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying WCET analysis by code transformations. In *4th Intl. Workshop on Worst-Case Execution Time Analysis*, Catania, Italy, June 30 2004. Satellite Workshop of the 16th Euromicro Conference on Real-Time Systems.

[13] Chang Yun Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Journal of Real–Time Systems*, 5(1):31–62, 1993.

[14] Stefan M. Petters. *Worst-Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real-Time Computer Systems, Technische Universität München, Munich, Germany, September 2002.

[15] Sergio Vale Aguiar Campos and Edmund M. Clarke and Wilfredo R. Marrero and Marius Minea. Verus: A Tool for Quantitative Analysis of Finite-State Real-Time Systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 70–78, 1995.

[16] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around ll caches. *Real-Time Systems*, 17(2-3):209–233, 1999.

Using a Processor Emulator on a Microkernel-based Operating System

Hidenari Koshimae, Yuki Kinebuchi, Shuichi Oikawa*, Tatsuo Nakajima

Department of Computer Science
Waseda University
{hide, yukikine, tatsuo}@dcl.info.waseda.ac.jp

*Department of Computer Science
University of Tsukuba
shui@cs.tsukuba.ac.jp

Abstract

This paper presents an architecture that allows multiple isolated commodity operating systems to run concurrently on a microkernel-based operating system without restricting the CPU architecture of guest operating systems. There are several systems that can execute multiple guest operating systems by using a virtual machine monitor or a microkernel-based operating system. Almost all of them, however, require that the CPU architecture of a guest operating system and a host operating system are identical and the guest operating system needs to be modified. Our architecture does not require the both of them and the prototype system that we have developed achieved the execution of multiple guest operating systems such as Linux and FreeBSD compiled for several CPU architecture without the modification of them. Since it involved decreasing the performance of a guest operating system, we propose several techniques to improve the performance in this paper.

Future embedded system will become more complex, so our approach offers a new way to reuse existing applications and operating systems.

1. Introduction

There are several systems providing virtualization that allows multiple guest operating systems to run concurrently. For example, Xen [1], a virtual machine monitor [10] that runs in a privileged CPU mode, supports to execute multiple commodity operating systems efficiently and isolate resources. In our design, we use a microkernel [9] instead of a virtual machine monitor. The microkernel also runs in a privileged CPU mode and provides only the minimum functionality in a kernel.

Usually, the modification of operating system kernels is required to execute guest operating systems on top of a microkernel. Xen also requires the modification of guest operating systems running on top of it without the virtualization support of each processor, such as Intel Virtualization Technology [6, 12]. However, our approach allows unmodified operating systems to run concurrently on top of a microkernel without processor supports by using a processor emulator. Our prototype system that we have developed allows unmodified commodity operating systems, such as Linux and Free BSD, compiled for x86, ARM, PowerPC and SPARC run concurrently and isolate the execution of respective operating systems for increasing security and reliability.

We have adopted the L4 microkernel [13] as the basis of the system and QEMU [2] as a processor emulator. The L4 microkernel itself provides only the minimum functionality of a kernel such as memory management, task management and inter-process communication primitives. We have ported QEMU, that was originally running on Linux and Windows, on the L4 microkernel. QEMU is able to emulate several CPU architectures, so our prototype system using QEMU can execute various operating systems that are compiled for several CPU architectures. A CPU architecture of guest operating system and that of host operating system on which guest operating system is actually executed may be different. This is impossible in Xen.

Using a processor emulator can execute various unmodified guest operating systems, but it imposes larger overhead compared with Xen and L4Linux (Linux running on top of the L4 microkernel) [4, 5] that run para-virtualized guest operating systems. In this paper, we present new approaches using *function-level dynamic translation* and *L4 memory map functionality* in order to reduce the overhead and improve the performance of guest operating systems.

The remainder of this paper is organized into 4 sections. Section 2 introduces a couple of related work. Section 3 describes the design and implementation of our prototype system. Section 4 proposes future work in order to improve the performance of guest operating systems, and Section concludes the paper. The discussion about usages and some performance results of our prototype system is described in another paper [7].

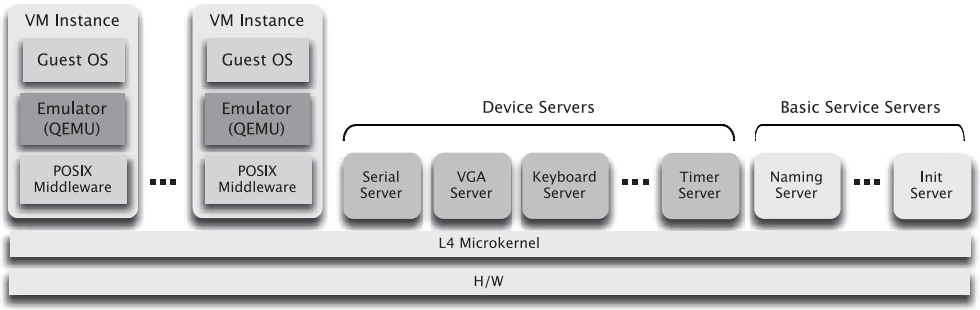


Figure 1. Overview of system architecture

2. Related Work

2.1. Xen

Xen is a virtual machine monitor for x86 that supports the execution of multiple guest operating systems with high levels of performance and resource isolation. Xen allows different commodity operating systems, such as Linux and Windows XP, to run concurrently and share hardware devices with safe hardware interface [3].

Xen supports unmodified applications, but requires kernels running on top of it to be modified. But, with a virtualization support by a processor, such as Intel Virtualization Technology or AMD Pacifica, Xen can also execute unmodified kernels.

2.2. L4Linux

L4Linux is a port of the Linux kernel to the L4 microkernel and the Linux kernel works as the L4Linux server. The applications of L4Linux run as user threads on top of the L4 microkernel side-by-side with other microkernel applications such as real-time components. Every L4Linux process has the same priority as other processes and is scheduled by the scheduler of the L4 microkernel.

2.3. Bochs

Bochs [8] is a portable x86 PC emulator. It is written in C++, and runs on most popular platforms such as x86, PowerPC, Alpha, SPARC and MIPS. Bochs allows many guest operating systems including Windows XP and Linux to run on itself. Bochs is mostly used for the operating system development; when a guest operating system crashes, it does not crash the host operating system, so the guest operating system can be debugged. Bochs interprets each of guest instructions one by one that enables detailed analysis of a guest operating system.

2.4. VMware Workstation

VMware Workstation [11] is a virtual machine monitor that can run unmodified operating systems built for the x86 architecture. It appears like an application running on commodity operating systems such as Linux and Windows. It installs VMM running in the privileged level as a device driver in order to use privileged level instructions. Since the implementation of the device driver depends on the assumed architecture, its portability is low.

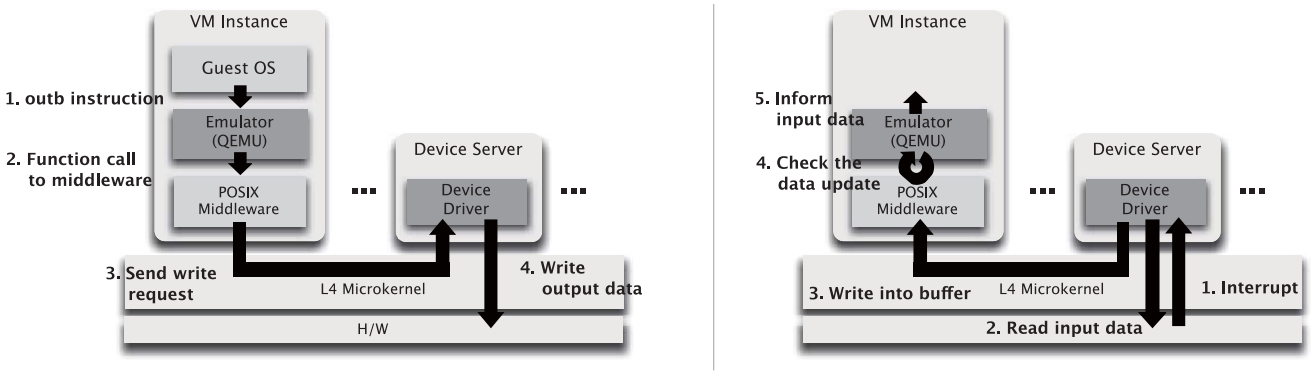
3. Design and Implementation

As described in Section 1, our prototype system is implemented with the L4 microkernel and QEMU. Figure 1 illustrates the overview of the system architecture of our prototype. Guest operating systems are executed as each VM instance whose single user thread contains a guest operating system, QEMU and POSIX middleware.

3.1. L4 Microkernel

The L4 microkernel is one of implementations for microkernel-based operating systems. The L4 microkernel itself provides only the minimum functionality of a kernel such as memory management, task management and interprocess communication primitives. In this system, only the L4 microkernel runs in privileged CPU mode and other programs run as non-privileged user threads. The L4 microkernel provides fast synchronous inter-process communication (IPC), and the communications between threads are performed by this IPC.

There are some versions of L4 microkernel and we have adopted L4Ka::Pistachio among them. In our prototype system, we use not only L4Ka::Pistachio but also Iguana that is designed as a base for the provision of operating system services and works on the L4Ka::Pistachio.



a. Device output process **b. Device input process**

Figure 2. Device I/O with device server

3.2. QEMU

QEMU is a processor emulator that runs on several commodity operating systems such as Linux or Windows. QEMU provides two kinds of emulation mode: the user mode emulation and the full system emulation. We used the latter emulation mode. The full system emulation that emulate a processor completely including MMU (Memory Management Unit) and interrupt processing contains the processor emulation of various CPU architectures. Latest version of QEMU supports full system emulation of x86, x86-64bit, ARM, PowerPC, SPARC and MIPS.

QEMU uses the technique called *dynamic translation* to execute unmodified guest operating systems on a host operating system. This translates the code of a guest operating system into the operations that are executable on a host operating system even if the CPU architecture of the guest operating system and the host operating system are different. However, this dynamic translation imposes large overhead, so QEMU contains a mechanism to reduce the overhead of dynamic translation to improve the performance of guest operating systems. QEMU translates the code of a guest operating system and executes the generated code by dividing into a block called a *translated block*. QEMU maintains a cache called *translated cache* that holds the most recently used translated blocks. When QEMU encounters the code that has been already translated and the target translated block exists in the translated cache, QEMU does not translate it again and uses the cached translated block; QEMU skips dynamic translation and eliminates the overhead of dynamic translation.

3.3. POSIX Middleware

We have developed POSIX middleware for porting QEMU to the L4 microkernel from Linux easily. QEMU used POSIX functions on Linux, so it is necessary to provide those functions on the L4 microkernel. Therefore we have implemented them as POSIX middleware. POSIX middleware also contains other functions required to run QEMU on the L4 microkernel.

There are some virtual devices that QEMU provides but we have not supported yet, so we modified a part of the source code of QEMU. In our design, however, we succeeded to run QEMU on the L4 microkernel without other modifications of QEMU.

3.4. Device Access

3.4.1 Device Server

We have implemented device servers that work as user threads on the L4 microkernel. The device servers are provided for every hardware device that our prototype supports. Each device server contains a device driver and manages device I/O to each hardware devices (User threads on the L4 microkernel have permissions to access hardware devices directly). Each device server is isolated from other components of the system, therefore device driver fault does not spread to the entire system. This is the advantage of component-based system using device servers.

Guest operating systems can share a hardware device with others by using a device server. This is because each device server mediates device I/O to/from multiple guest operating systems. For example, when two or more Linux instances are executed as guest operating systems, the user can input the data through a single keyboard to arbitrary Linux by switching the input destination dynamically. In the case of VGA console, a user can display the output of arbitrary Linux on a screen and switch respective guest Linux instances, or the user can display all outputs of guest Linux instances by dividing the screen.

The device outputs from guest operating systems are performed as following (see also Figure 2-a). When a guest operating system issues a device output instruction, such as `outb` in x86, then QEMU converts it into a function call to POSIX middleware. Then, POSIX middleware transmits the request of data output to a device server by using the synchronous IPC. Finally, the device server performs the device I/O by using device driver contained in itself.

The device inputs to guest operating systems are performed as following (see also Figure 2-b). When the device server receives an interrupt, it reads input data from a hardware device. Then, the device server writes input data into a circular buffer that POSIX middleware uses in order to acquire input data from the device server. QEMU monitors whether there is a data update in the circular buffer by polling it. If there is any update, QEMU reads the input data from the buffer, and transfers them to a target guest operating system.

3.4.2 Internal Device Driver

Another way of device I/O in our prototype is to use internal device drivers that are included in POSIX middleware. VM instances, as well as device servers, have a permission to access hardware devices directly, so they are able to perform device I/O by themselves. This approach, however, prevents multiple guest operating systems from sharing hardware devices. There are several ways that allow device sharing, for example, particular VM instance serves all device I/O to a particular hardware device for all other VM instances, or each VM instance cooperates with others to perform device I/O by using their own internal device drivers. However, these require additional mechanisms to be added in our prototype.

This device I/O model does not require synchronous communications with device servers, therefore faster I/O can be achieved than using device servers. So, our prototype uses internal device drivers for the output to VGA graphics and a VM instance writes output data into VRAM directly.

4. Efforts for Performance Improvement

It is advantageous to use a processor emulator to execute guest operating systems without the modification of a guest operating system and the availability of various commodity operating systems. However, as compared with Xen or L4Linux, it causes larger overheads.

In this section, we describe the future work to eliminate the overhead and to improve the performance of a guest operating.

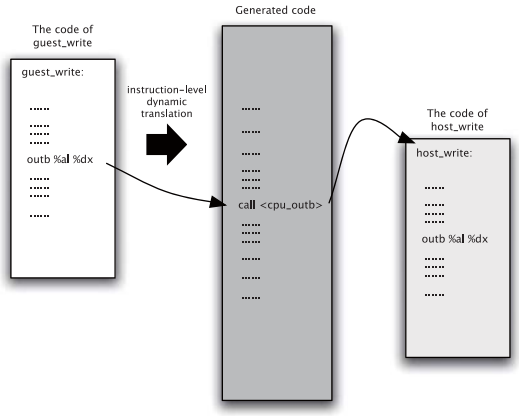


Figure 3. Function-level Dynamic Translation

4.1. Function–level Dynamic Translation

We believe that the overhead of dynamic translation is a main reason in the performance decrease. Actually, when we restarted Linux on our prototype system immediately after Linux was started first, the start up time of the second execution was shortened to about 1/3 compared with the first execution. We consider that this is because the mechanism for the performance improvement of QEMU (described at 3.2) worked effectively; the efficient use of translated blocks preserved in the translated cache has reduced the frequency of dynamic translations. We believe, therefore, the improvement of the performance of a guest operating system is achieved by reducing the frequency of dynamic translations. In this paper, we propose a technique to improve performance called *function-level dynamic translation*.

When a guest operating system is executed by QEMU, QEMU translates each instruction of the guest operating system code into the operations that are executable on a host operating system (we call this dynamic translation as *instruction-level dynamic translation*). Figure 3 illustrates an example of this. A guest operating system and a host operating system have the similar functions, `guest_write` and `host_write`, that write data into a certain hardware device. When a guest operating system attempts to write data into the device with `guest_write`, QEMU translates the code of `guest_write` and executes generated codes. For example, when `guest_write` outputs the data by `outb` instruction of x86, QEMU converts this instruction into a function call for `cpu_outb`, and this `cpu_outb` finally executes `host_write`; thus, `guest_write` is performed by `host_write` eventually.

On the other hand, Figure 4 illustrates an example using function-level dynamic translation. Here, we use `fast_guest_write` function containing only a few in-

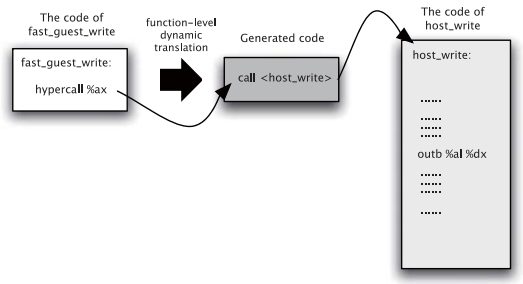


Figure 4. Execution flow of fast_guest_write function with function-level dynamic translation

structions including a `hypercall` instruction instead of the `guest_write` function. This `hypercall` instruction is virtual instruction that we will implement by extending the instruction set that QEMU supports. QEMU converts the `hypercall` instruction into the function call to `host_write` directly and executes it. Thus, function-level dynamic translation allows a guest operating system to call an arbitrary function of a host operating system by using the `hypercall` instruction with a little overhead of dynamic translation.

When Figure 3 and Figure 4 are compared, it is obvious that the latter case is more efficient. In the former case, the number of instructions that are required to be translated dynamically is the same number of instructions contained in the `guest_write` function. In the latter case, however, only a few dynamic translations are required and the generated code is very compact and efficient.

Using function-level dynamic translation requires the modification of guest operating system. However, it is possible to realize the optimization without the modification of the kernel. The modification can be contained into device drivers that can be offered by kernel modules. This means that our approach can be used only by modifying kernel modules.

4.2. Using L4 Memory Map Function

Since many modern processors has MMU, virtual machines and emulators need to offer the MMU functionality. Commodity operating systems usually do not provide the function to control memory space for a process running on top of it, so virtual machines and emulators should provide the MMU functionality by themselves. Mainly there are two ways to implement it. One does an entire memory address translation with software that is employed by QEMU and Bochs. The other installs a special device driver to the host operating system to allow a user-level program to control page tables directly, which is employed by VMware Workstation.

QEMU provides *software MMU* that emulates MMU only with a software function. When a guest program, a guest kernel or a user-level guest program, attempts to access memory, software MMU look up the corresponding page table entry in the page tables constructed by the guest kernel and calculates the physical address from the virtual address. Figure 5 shows the mechanism of software MMU. For simplicity it assumes that the page tables map virtual addresses and physical addresses one-by-one. The calculated physical address is used as the offset for the memory region allocated for the guest operating system. In this way, every time guest programs access memory, software MMU performs the page table lookup. Therefore, a single memory access expands to multiple memory accesses including the accesses to the page tables. The resulting time for the memory access would be factor of ten.

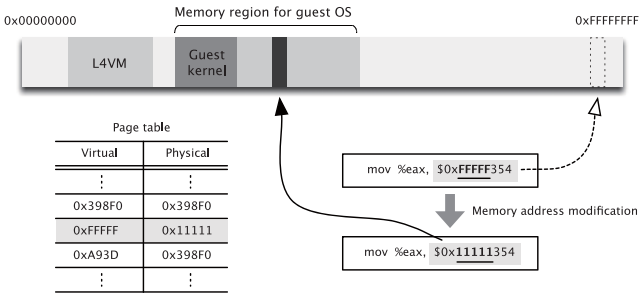


Figure 5. Dynamic memory address modification with software MMU

Unlike commodity operating systems, the L4 microkernel provides APIs to manage address space. Therefore, user threads on the L4 microkernel can create new address spaces and map a memory section into them. We propose another method of MMU emulation that employs this L4 memory map function. In this method, QEMU creates and maps memory regions into the *external address space* according to the page tables constructed by the guest kernel (see Figure 6). Each guest program is executed in a separate address space, so it can access memory directly without the interposition of software MMU. We suppose this eliminates the overhead of software MMU and achieves the performance improvement.

Furthermore, we suppose that not only software MMU but also dynamic translation of user-level guest programs could be unnecessary when the guest CPU architecture and the host CPU architecture are identical. For this, we need additional modifications so that traps and page faults performed by the user-level programs can be transfered to QEMU and the guest kernel within QEMU can handle them properly. Since a user-level guest program and a guest ker-

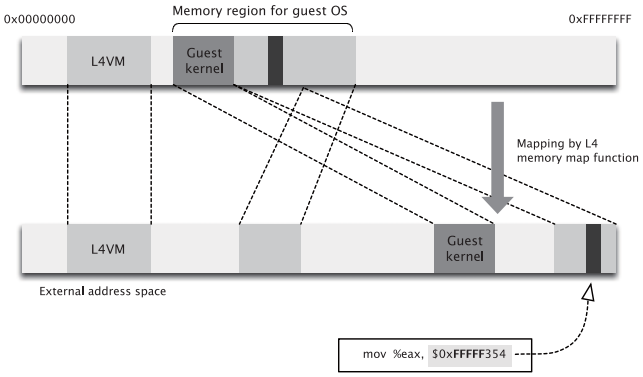


Figure 6. L4 memory map function

nel are executed in a separate address space, the guest program cannot access the kernel. In this model, only the guest kernel code is translated. Therefore, we suppose that the overhead of software MMU and dynamic translation are decreased and the performance improvement can be achieved.

4.3. Efficient Use of Translated Cache

As described in Section 3, QEMU skips dynamic translation and enhances the performance if translated blocks are in the translated cache. The size of this cache to preserve translated blocks is limited, so it should be managed efficiently. The latest version of the QEMU implementation, however, flushes all translated blocks when the cache is filled. We believe the introduction of efficient management mechanism for the translated cache also improves the performance well.

5. Conclusion

Using a processor emulator on a microkernel-based operating system allows various unmodified commodity operating systems to run concurrently on tops of the microkernel-based operating system. The prototype system that we have implemented achieved to execute two or more Linux and FreeBSD concurrently, that are compiled for different CPU architectures on top of the L4 microkernel, and allows these operating systems to share a single hardware device safely by using device servers. Our prototype has showed that the emulation of a processor introduces very large overheads while there are a lot of advantages. We believe, however, using techniques for the performance improvement that we have proposed in this paper, such as function-level dynamic translation, improves the performance of guest operating systems.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003. University of Cambridge Computer Laboratory, ACM Press.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, June 2005.

[3] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Suport for the On-Demand IT Infrastructure*, Boston, MA, October 2004. University of Cambridge Computer Laboratory.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, Saint Malo, France, 1997. ACM Press.

[5] H. Härtig, M. Hohmuth, and J. Wolter. Taming linux. In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*, Adelaide, Australia, September 1998. IEEE Press.

[6] Intel Corporation. Intel virtualization technology. `http://www.intel.com/technology/computing/vptech/`.

[7] Y. Kinebuchi, H. Koshimae, S. Oikawa, and T. Nakajima. Virtualization techniques for embedded systems. In *Proceedings of the Work-in-Progress Session: the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[8] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Proceedings of the Linux J.*, 1996(29es):7, 1996.

[9] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, Colorado, United States, December 1995. ACM Press.

[10] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[11] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[12] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[13] University of Karlsruhe, System Architecture Group. L4ka::pistachio microkernel. `http://l4ka.org/projects/pistachio/`.

Virtualization Techniques for Embedded Systems

Yuki Kinebuchi, Hidenari Koshimae, Shuichi Oikawa*, Tatsuo Nakajima
Department of Computer Science
Waseda University
{yukikine, hide, tatsuo}@dcl.info.waseda.ac.jp

*Department of Computer Science
University of Tsukuba
shui@cs.tsukuba.ac.jp

Abstract

Like existing commodity computer systems, embedded systems are no more small, static, stable nor safe. They have huge amount of source codes, interact with the internet, download and install applications from external, which incur the increase of bugs and security holes. To solve these problems, we propose to leverage virtualization techniques on embedded systems. We ported a machine emulator supporting various architectures to run on a microkernel that runs on several architectures.

In this paper, we discuss benefits of using virtualization techniques on embedded system, introduce the implementation and the evaluation of a machine emulator running on a microkernel, and discuss some issues using our system on embedded systems.

1. Introduction

Today, 'embedded systems' does not only mean control systems used in automobiles and factories, but it also includes mobile phones and car navigation systems, with luxurious applications such as games and multimedia. Unlike the traditional embedded systems, these systems share some problems with existing commodity systems, such as increasing bugs and security holes.

As the embedded systems extend their functionality, their code size have drastically increased. The system of a single mobile phone is made up of more than a million lines of code, which is close to the size of commodity systems. Since a large software entail many bugs, it requires large time of testing and debugging, which consumes more time and human resources.

In addition, since modern embedded systems have a capability to access the internet, they are now facing to security risks to be attacked through the internet and installing malicious applications such as viruses and trojan codes. As shown in existing commodity operating systems, security holes have never gone. This means that same thing will happen in embedded systems too. Since embedded systems have less chance to update their codes, their security hole would remain for suf cient time to be attacked.

Virtualization techniques have been researched to increase the system security, reuse of traditional operating systems and applications, and increase the system throughput, but mainly for commodity systems. We propose to leverage virtualization techniques on embedded systems to x some common problems for embedded systems and commodity systems, and also some problems speci c to embedded systems.

We constructed machine emulator running on microkernel, to achieve the isolation of untrusted application and reuse of existing softwares without any modi cation. In this paper, we propose some uses of machine emulator running on a microkernel, evaluate the performance, and discuss some issues.

The next section introduces some related work. Section 3 discusses some bene ts of using virtual machine monitors on embedded systems. Section 4 introduces the implementation of a machine emulator running on a microkernel. Section 5 proposes some applications using our system. Section 6 introduces the result of evaluations. Section 7 discusses some issues to use our system on embedded systems. Finally the section 8 concludes the paper.

2. Related Work

In this section, we introduce some existing virtualization systems.

Bochs[9] is a machine emulator which has a capability to run guest operating systems built for the x86 architecture without modi cations. The purpose of Bochs seems to be an accurate emulation of the x86 architecture to support a detailed debugging of system level applications running on top of it. Since the performance is lower than the other virtualization techniques, it is not suitable for running a guest operating system for actual use.

Xen[1] is a virtual machine monitor leveraging a virtualization technique called para-virtualization, which requires

the modi cation of guest operating systems to be run on it. The purpose of Xen is to run multiple servers simultaneously on a single computer, in order to reduce the number of machines and increase the throughput of whole system. Since the implementation of Xen highly depends on the x86 architecture, it is not ported to embedded systems.

VMware Workstation[12] is a virtual machine monitor that can run unmodi ed operating systems built for the x86 architecture. It runs as an application running on commodity operating systems such as Linux and Windows; however, it installs virtual machine monitor running in the privileged level as a device driver in order to use privileged level instructions. This is to increase the performance of running guest operating systems. It is used to run another operating system on a commodity operating system to use operating system speci c applications and debugging system level programs.

L4Linux[6] is a para-virtualized Linux kernel running on top of L4. The purpose of L4Linux is to reconstruct the existing monolithic operating systems to gain e xibility and extensibility. Wombat[5] is also an para-virtualized Linux kernel that runs on the L4-embedded kernel[4] and Iguana[3]. Iguana is a platform constructed on top of the L4-embedded kernel to provide the basic operating system functionality over the embedded systems. Wombat enables the use of Linux applications on L4-embedded systems.

Pre-virtualization[10] is a technique that automatically replaces virtualization-sensitive instructions with hypervisor calls that invoke functions emulating the original instructions. The purpose of pre-virtualization is to reduce the cost of modifying guest operating system kernel in para-virtualization, but keep the performance. The technique also could be applied to embedded systems by modifying a compiler for a target architecture.

3. Benefits of Using Virtual Machine Monitors on Embedded Systems

In this section, we discuss some bene ts of using virtualization techniques on embedded systems to advance system security, software reuse, support running multiple operating system simultaneously, and improving debugging functionality.

3.1. Security

Since some modern embedded systems have capability to access the internet, there is a risk to be attacked by remote hosts. For example, when a browser has a stack over o w vulnerability, they can be forced to run an arbitrary code by a malicious remote host.

In addition, some embedded systems can dynamically change the system construction by adding and removing applications. For example, mobile phones can extend their functionality by downloading and installing a new application through the internet. Although, the dynamical reconstruction of a system incur a risk of being affected by viruses and trojan codes.

To reduce such security risks, system isolation using a virtual machine monitor is proposed. It splits the system to run on different operating systems. Untrusted applications download from external, runs on a operating system which is isolated by the main operating system. When untrusted applications intrude the operating system, a damage in contained in isolated operating system, and does not affect the main system.

3.2. Software Reuse

Even the size of embedded softwares increases, development time would not change or even getting shorter. In order to reduce the development time, operating systems and applications must be reused.

One of the bene ts of using virtual machines on embedded systems is to provide an interface compatible with old hardwares. By using virtual machine, not only applications but also operating systems can be reused.

3.3. Simultaneous Multiple Operating Systems

Modern embedded systems have to support the functionality of both embedded systems and commodity systems. For example, real-time control applications and time sharing applications must run simultaneously in a single embedded system. Achieving both real-time support and luxurious functions with a single operating system is rather challenging. Instead of supporting both of them, running different operating system simultaneously on virtual machine monitor will be more simple to solve the problem.

3.4. Debugging and Monitoring

Operating systems running on embedded systems are dif cult to be debugged. It is done by modifying the operating system to interact with the debugger running on a host machine, or using a hardware tools such as JTAG and ICE. The former cannot be used when the operating system itself is crashed, and the latter is not supported by every hardwares.

By running operating systems on a virtualized environment, the debugging and monitoring could be performed by the underlying virtual machines. It does not require the modi cation of guest operating systems nor special hardware.

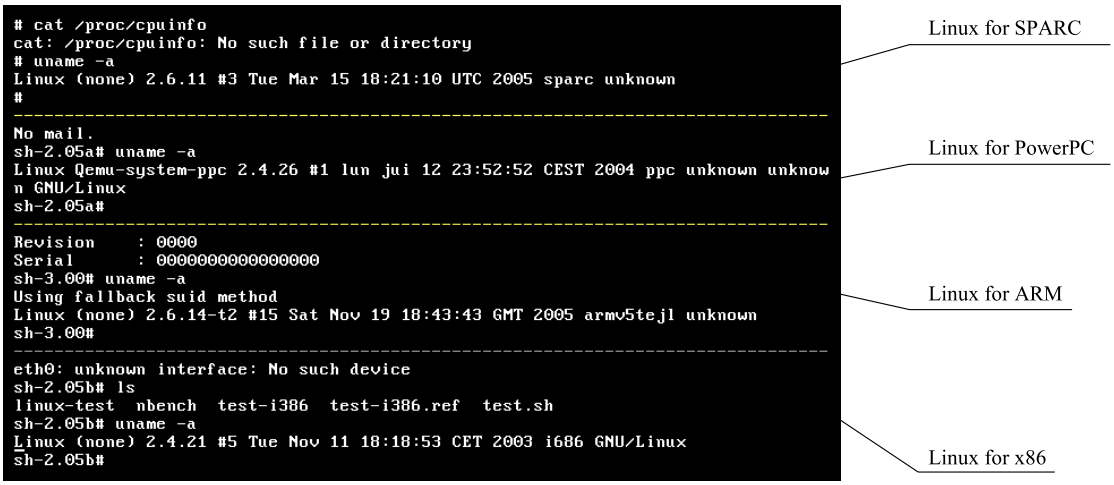


Figure 1. Four Linux for SPARC, PowerPC, ARM and x86 architecture on L4 (Screenshot)

4. QEMU on L4

We ported the QEMU machine emulator[2] to the L4Ka::Pistachio microkernel[13] (L4 for short) to construct a virtual execution environment running on embedded systems. QEMU is a dynamic translator supporting various different architectures for the host and the guest. When the host and the guest architecture is different, it transfers the code and keep them in a buffer to reuse it when the same code is executed again. We adopted Iguana, the environment which helps the construction of application on L4, to build our system. The detail of the implementation is described in another paper [8].

5. Applications Using QEMU on L4

5.1. Emulating Multiple Architectures

The primary use of our machine emulator is the reuse of existing operating systems on top of various types of architectures. Since QEMU exposes an interface compatible with existing hardwares, guest operating systems can run on top of it without any modi cations. QEMU can run a guest operating system even if the host and the guest architecture are different. For example, as shown in Figure 2, we constructed a system running four Linux for SPARC, PowerPC, ARM and x86 on top of L4 simultaneously. Figure 1 shows the screenshot of running the systems. The console server splits the monitor into four parts and makes each guest Linux to use one of them.

Using our machine emulator enables to use unmodi ed operating systems whose architectures are different from the host architecture. This also means that any modi cations are not required for any guest applications. The reuse of guest applications is as bene cial as that of guest operating systems. In embedded systems, the number of available applications is relatively fewer; for example, the number of applications that run on ARM processors, that can be used for mobile phones, is less than that on x86 processors. However, if such applications on x86 processors are also available on ARM processors, more services might be realized on embedded systems with the minimal effort.

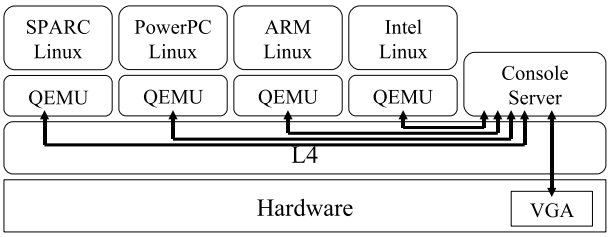


Figure 2. Four Linux for SPARC, PowerPC, ARM and x86 architecture on L4

5.2. Debugging Supports

One of the drawbacks of L4 is the lack of convenient debugging environments. L4 provides KDB, a kernel embedded debugger, which provides some functions for debugging applications running on L4. Although, the debugging functions are primitive, it is not convenient enough for a practical debugging.

We have proposed another use of our machine emulator as a debugger for operating systems and L4 application. QEMU includes a GDB stub within it. By connecting the GDB stub with GDB running on top of a host operating system, users can analyze programs running on QEMU (Figure 3).

Original QEMU supports not only the full system emulation, but also the user mode emulation that has a capability to run and debug Linux applications. As the full system emulation, it translates an application code into TBs. The difference is that when a system call is executed by a guest application, QEMU interposes it and invokes a system call instead of the application itself. In Linux, for the x86 architecture, instructions for invoking a system call is `int 0x80`. In order to support the debugging of L4 applications, QEMU needs to be modi ed to interpret the L4 system call ABI.

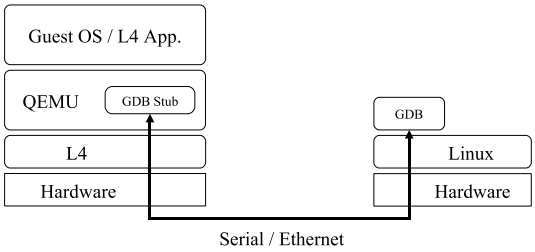


Figure 3. Debugging with QEMU

5.3. Anomaly Detection/Recovery

Alex Ho et al. proposed a taint-based protection using a machine emulator[7]. In normal times, an operating system runs on a virtual machine monitor. When the CPU is executing a code that interacts with data downloaded through the internet, the execution is dynamically switched on to the machine emulator. In this way, it reduces the performance degradation and protect the system from tainted data.

Using our system, we propose a similar system that offers an anomaly detection and a recovery (Figure 4). In normal times, applications run directly on the L4 microkernel. When the system nds the symptom of application anomaly, the application is migrated to run on QEMU. QEMU runs the application and analyzes it in detail. When it detects an anomaly, it stops the execution of the application, and if possible, it recovers the application and puts it back to run directly on the L4 microkernel again. In this way, the system can realize the anomaly detection and recovery with near-to-native performance.

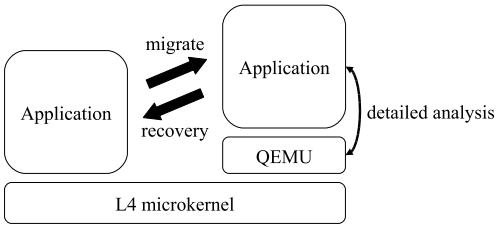


Figure 4. Anomaly Detection

6. Evaluation

In this section, we evaluate the performance of Linux running on QEMU on L4. We used LMbench[11] to measure their performance. LMbench is a cross platform benchmark to measure the performance of operating system primitives. We built LMbench for three different architectures; x86, SPARC and ARM. The measurements were performed on non-virtualized (native) Linux for x86 architecture, virtualized Linux for x86, ARM and SPARC. Non-virtualized and virtualized Linux for x86 are the same. The machine we used for measurement is IBM ThinkPad R40, with 1.3GHz CPU and 768MB memory. The SpeedStep(R) technology of CPU is disabled for accuracy.

LMbench leverages `gettimeofday()` to measure time. Although when invoking `gettimeofday()` on Linux running on QEMU, it does not return an accurate time. To let LMbench obtain accurate time, we modi ed QEMU to add an instruction to the instruction set of each target architecture. The instruction returns the time elapsed since the start of the host machine on a microsecond basis, which is calculated using host `rdtsc` instruction. The extended instruction is embedded in the code of LMbench with inline assembler by specifying the bit pattern of the extended instruction directly, like the following code.

```
asm("get_usec:");
asm("    .long 0xffffffff");
asm("    mov pc, lr"); // return

int func()
{
    unsiged int tt;
    tt = get_usec();
}
```

When measuring the performance of a dynamic translator, we have to consider that the performance of a guest application running on QEMU depends on whether the application is not yet translated, or it is already translated and saved on a buffer. Since LMbench 'warmup' before the real measurement, the code of LMbench will be translated before the execution. So the result here is the performance of application on buffer caches.

Table 1 shows the result of measurement. We measured the system call latency, context switch latency, the bandwidth of reading from and writing to the memory, and bandwidth of reading a le.

The rst row of the table shows the system call performance. Comparing non-virtualized and virtualized Linux for x86, the performance is decreased by a factor of 11.

The second row shows the latency of the context switch. The performance decreased by approximately a factor of 90.

	x86(Native)	x86	ARM	SPARC
lat_syscall (μsec)	0.2634	3.0967	18.9526	3.0504
lat_ctx (μsec)	0.54	48.10	80.13	83.09
bw_mem rd (MB/s)	9328.49	1051.80	618.39	508.50
bw_mem wr (MB/s)	5509.96	597.21	436.50	379.23
bw_ le _rd (MB/s)	1557.27	36.29	45.42	41.47

Table 1. LMbench measurement result

The third and forth row show the bandwidth of reading and writing a memory. The throughput decreased by a factor of 10. The memory access speed of programs running on QEMU is the one of the major overhead of QEMU, which we describe in more detail in Section 7.1.

The last row shows the bandwidth of reading a le. Comparing non-virtualized and virtualized Linux for x86, the performance is decreased by approximately a factor of 40.

7. Discussion

Running QEMU on the L4 microkernel realizes the running of multiple operating systems simultaneously on a single hardware, isolation between the guest operating systems with giving them the illusion of using a hardware by itself, and the reuse of operating systems without modi - cation even for the operating system for different architectures. However, as shown in Section 6, the performance of guest operating systems is low. In this section we propose a technique to improve the performance of dynamic translators on microkernels and discuss the real-time issue of our system.

7.1. Performance Improvement

As shown in Table 1, comparing virtualized Linux for x86 to non-virtualized Linux for x86, the bandwidth of memory is decreased by approximately a factor of 10. The decrease of memory bandwidth derives from the implementation of a software MMU, which emulates MMU functions such as an address translation and a memory protection with software implementation. When a virtualized program access a memory through MMU translation (which is ordinary in commodity operating systems), QEMU transfers the required virtual address to a physical address referring the pagetable on the target memory.

To improve the memory access performance, we propose to modify the software MMU to leverage the memory mapping function of underlying L4. When a guest operating system remaps the memory into memory space, QEMU uses the L4 memory mapping function to map memory into the guest address space. The guest operating system would be run in the address space, it does not need to translate a virtual address on every single memory access.

7.2. Real-time Issue

Embedded systems tend to have real-time requirements. Although, with using our system, it is hard to guarantee real-time scheduling. When executing a guest code with dynamic translator, rst the guest code is translated to host code, saved in a buffer, and then executed. When the buffer is full, the whole buffer it is ushed. The current implementation of QEMU does not support the detailed management of buffers. The aspect of the dynamic translator is unpredictable.

8. Conclusion

In this paper, we discussed some bene ts to use virtualization techniques on embedded systems to adjust some problems of modern embedded systems. We propose to use a dynamic translator on a microkernel, and evaluated the performance.

The system supports the isolated environment to achieve system security and reuse of operating systems and applications without any modi cations and running multiple commodity operating systems on a single hardware. However, its performance was not enough to be used on embedded systems which have strict resource requirements, and also lacking the support of real-time guarantee.

9. Future Direction

We are now working on the development of a new virtual machine monitor, which provides some functions to guarantee the real-time scheduling of applications running on para-virtualized guest operating systems.

To achieve real-time scheduling on virtualized environment, the guest operating system and the host virtual machine monitor should schedule applications in cooperative

behavior. This means that both the host and the guest must share their scheduling information. We are planning to modify a guest operating system, especially its scheduler, to invoke the scheduling functions provided by the host in order to map some real-time processes to the host real-time processes.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. War eld. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2005.

[3] Embedded, Real-Time, and Operating Systems. Iguana. `http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/`.

[4] Embedded, Real-Time, and Operating Systems. L4-embedded. `http://www.ertos.nicta.com.au/research/l4/embedded.pml`.

[5] Embedded, Real-Time, and Operating Systems. Wombat. `http://www.ertos.nicta.com.au/research/virtualisation/`.

[6] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, Oct. 5–8 1997.

[7] A. Ho, M. Fetterman, C. Clark, A. War eld, and S. Hand. Practical taint-based protection using demand emulation and intel research cambridge. In *Proceedings of the EuroSys 2006*, Leuven, Belgium, Apr. 18–21 2006.

[8] H. Koshimae, Y. Kinebuchi, S. Oikawa, and T. Nakajima. Using a processor emulator on a microkernel-based operating system. In *Proceedings of the Work-in-Progress Session: the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 16–18 2006.

[9] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es):7, 1996.

[10] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), Nov. 2005.

[11] L. McVoy and C. Staelin. Lmbench - tools for performance analysis. `http://www.bitmover.com/lmbench/`.

[12] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[13] System Architecture Group. L4ka::pistachio microkernel. `http://l4ka.org/projects/pistachio/`.