A TOOL CHAIN FOR THE SFQ-LSRDP COMPUTER

Farhad Mehdipour, Hiroaki Honda, Hiroshi Kataoka, Koji Inoue, Kazuaki Murakami
Kyushu University, Japan

CREST-ULP SFQ-LSRDP PROJECT

Technical Report

November 2011
Summary of Tool Chain

The main objective of the tool chain is to generate a configuration bit-stream file (green path in below figure) for the LSRDP and an executable code for the GPP (blue path in the following figure). The input to the tool chain is an application C code. This code is analyzed and the critical sections are extracted in the form of data flow graphs from the application code. According to the green path, DFGs are mapped onto the LSRDP through locating DFG nodes on the PEs, routing interconnections as well as positioning input/output nodes on the proper I/O ports. Considering the LSRDP architectural specifications in the mapping process is a basic requirement. Configurations’ bit-stream corresponding to each one of DFGs can be generated after completion of the mapping stage. Referring to the blue path, firstly the input code is modified so that the special and predefined instructions for communicating to the LSRDP hardware are inserted in the code and the parts of code corresponding to the critical sections are eliminated. The modified code is compiled using COINS as a compiler infrastructure for the target GPP. Final executable code and the configuration files can be used as an input to a simulator which is employed for performance evaluations. A part of compiler tools can be customized for utilizing in the LSRDP design phase as well.
1. Introduction

In order to achieve a high-performance and low-power consuming computing system, a single-flux quantum based micro accelerator (referred as large-scale reconfigurable data-path or LSRDP) has been proposed [20] which is augmented to a CMOS general purpose processor, a memory (Fig. 1). The proposed architecture is expected to be a 10TFLOPS desk-side computer with low electric power consumption and it is suitable for execution of the scientific applications demanding massive computations. LSRDP utilizes a data-path comprising reconfigurable interconnections to connect several floating-point processing units together. The SFQ-RDP is implemented as a two-dimensional array of processing elements (PEs) such that one PE can be connected through Operand Routing Networks (ORNs) to one or more PEs in the next row.

The main task of LSRDP is executing the most frequently executed portions of applications or in other word the part of applications demanding massive and time-consuming computations. In a general view, critical segments are pulled-out from applications and their corresponding configuration bit-streams are generated. During execution of the application on the base processor, related configurations associated with critical segments are loaded on LSRDP and executed to achieve higher performance and lower power consumption.

![Fig. 1. Overall architecture of the SFQ-LSRD computer](image-url)
Developing necessary tools for compiling applications, generating DFGs and their configuration bit-streams are essential phases of developing target computer which will be discussed in the following sections. It should be noted that we rely on our previous works such as [11][12][14] in developing the architecture and tool chain for the SFQ-LSRDP as well.

Fig. 1 displays an overall architecture of a high performance computer consisting of a GPP, LSRDP as an accelerator and main memory which are all connected through a shared bus to each other. Generally, the LSRDP is constructed as a two-dimensional array of PEs communicating together via ORNs, streaming buffer (SB) which is the FIFO type buffer and Streaming Memory Access Controller (SMAC) for managing I/O data. Each PE can be fed through SMAC and SB and the resultant of each PE can be transferred to one or more PEs via ORN switches. Reverse data flow connections are not supported, which means that the flow of data in the array is only in one direction. The LSRDP should be an adaptable accelerator, because it is aimed to target various scientific applications. In order to satisfy this requirement, the architecture is featured with dynamically reconfigurable PEs and ORNs. Originally, an ORN consists of programmable switches. By means of configuring control signals provided with PEs and ORN switches, the function of LSRDP can be determined at run time. Such flexibility makes it possible to implement various DFGs on the array.

A data flow graph (DFG) extracted from a target application program is mapped onto the LSRDP array. Since the cascaded PEs can generate a final result without temporally memorizing intermediate data, the number of memory load/store operations corresponding to spill codes can be reduced. Therefore, memory bandwidth required to achieve a high performance can be reduced. Furthermore, since a loop-body mapped into the PE array is executed in a pipeline fashion, LSRDP can provide a high computing throughput.

Fig. 2 shows a detailed view of the proposed LSRDP architecture. Layout of LSRDP ensembles a checker layout in which any PE can implement only ADD/SUB or MUL (Fig. 2(b)). The PE architecture (Fig. 2(c)) has one functional unit (FU) and a transfer unit, however, there is a possibility of implementing two simultaneous TUs by the FU (totally three TUs). An additional mux should be used inside the PE to choose between FU’s output and the input. It is assumed that FUs can implement basic 64-bit double-precision floating point operations such as ADD, SUB and MUL. Input/outputs ports are located at the top and bottom boundaries of the LSRDP architecture. PEs of each row are connected to the PEs in the next row through...
ORNs as routing resources. Fig. 2(d) shows the definition of the connection length and the maximum connection length (MCL) on a piece of LSRDP architecture. It can be seen that the connection length of two PEs is the horizontal distance of the PEs. Correspondingly, the MCL size is the maximum horizontal distance of two PEs located in two subsequent rows (Fig. 2(e)). ORNs’ functionality is similar to a multiplexer however; ORNs are composed of cross-bar switches (CBs) (Fig. 2(c)). Similar to other components of the LSRDP, CBs are also implemented by means of Josephson Junctions as the basic elements of the SFQ circuits [8].

Immediate (constant) operands are implemented by means of 64-bit immediate registers located in each PE. Constant values are transferred to the immediate registers within configuration phase through a serial bit-stream. LSRDP is a reconfigurable hardware that can be configured within run-time using the bit-stream generated for DFGs. Upon reaching to a critical segment during application execution, a reconfiguration
phase starts and the LSRDP configurable architecture including ORN, immediate registers and PEs are reconfigured. Further, it is assumed that sixteen memory modules of 1800 Mbps/pin are used in the memory array. Each memory module uses one channel for input and the other channel for output. Data bus width for transferring data is 64 bit where double-precision data (8 bytes) is handled in the computations. Therefore, the data transfer rate is almost 24GB/s.

Table 1 summarizes the detailed specification of the LSRDP which has been determined throughout a design procedure [11]. More details on the architecture design procedure can be found in [10] and [11]. The main focus of this report on the tool chain implemented for the LSRDP. A designer will be able to develop a target application by means of this tool chain. In the following sections a detailed overview on the proposed tool flow will be explained.

2. LSRDP Tool Chain

2.1. Overview

The main objective of the tool chain is to generate a configuration bit-stream file (blue path on Fig. 3) for the LSRDP and an executable code for the GPP (green path on Fig. 3). The input to the tool chain is an application C code. This code is analyzed and the critical sections are extracted in the form of data flow graphs from the application code.

Generally, extracting data flow graphs from applications can be performed manually or automatically by means of a sophisticated high-level profiling tool. In the former case, programmer needs to have a sufficient knowledge on the application and its detailed characteristics. On the other hand, automation of the hw/sw co-design methodology [5] brings with it the need to develop sophisticated high-level profiling tools e.g. gprof[2], HALT [21], ProfileME [4].

According to the green path, DFGs are mapped onto the LSRDP through locating DFG nodes on the PEs, routing interconnections as well as positioning input/output nodes on the proper I/O ports. Considering the LSRDP architectural specifications in the mapping process is a basic requirement. Configurations’ bit-stream corresponding to each one of DFGs can be generated after completion of the mapping stage.

Referring to the blue path, firstly the original code is modified so that the special and predefined instructions for communicating to the LSRDP hardware are inserted in the code and the parts of code corresponding to the critical sections are eliminated. The modified code is compiled using COINS [3] as a compiler infrastructure for the target GPP. Final executable code and the configuration files can be used as an input to a simulator which is employed for performance evaluations. A part of compiler tools can be customized for utilizing in the LSRDP design phase as well.

It should be noted that the proposed tool chain can be also applicable to the architectures which are implemented using CMOS technology. The main difference is in the DFG placement and routing tool which is implicitly or explicitly considering the constraints and limitations originating from SFQ implementation. Therefore, this part of tool flow is customized based on the SFQ technology’s features, however it is also applicable to CMOS architectures with slight modifications.

![Fig. 3. The tool chain](image_url)
2.2. DFG Extraction

Acceleration of the algorithm can be achieved by implementing critical sections on dedicated hardware accelerators. In this approach, one important step is to isolating critical and non-critical parts of program for executing them on the hardware and base processor, respectively. Critical portions of a program represent the highly executed or hot portions of it which are suitable to execute on the hardware to gain a higher performance. In contrast, non-critical portions have no high execution frequency and do not worth to be run on the base processor. Program execution tends to spend most of the time in a small fraction of code (90% of the execution time comes from 10% of the code). Almost 85% of critical regions are in inner loops and 15% remaining regions are in functions. Partitioning process can automatically isolate critical and non-critical segments of the application code. After pulling the critical segments out of application code, data flow graphs corresponding to them are generated. Data flow graph is used for converting hot portions to configuration bit-stream.

Detecting critical portions in an application code can be done at high level or binary level of the code. The latter case is done on the primitive instructions of the base processor. Using binary level brings about more transparency, more accurate software estimation and similar speedup to source-level for numerous applications while keeping the binary compatibility. Profilers like Dynamo, Daisy and Trnasmeta’s code Morphing software optimize and/or translate binaries to better suite the underlying hardware. In former case, application code in its high level description is analyzed to detect and pull out the hot portions. The main focus is on the most critical software loops. In this case, a compiler infrastructure can be used. Different levels of representations can be obtained and various analysis (control and data flow analysis) and optimizations can be done using a compiler infrastructure. SUIF [19] and COINS [3] are two instances of the compiler infrastructures. The former is based on c and the latter is java based. The COINS compiler infrastructure is composed of three major parts:

1. The front-end part translates the source program into HIR (High-level Intermediate Representation).
2. The middle part converts HIR into LIR (Low-level Intermediate Representation) and also does optimization and parallelization transformations on HIR or LIR.
3. The back-end part generates the assembler code of the target machine from LIR.

Also, it includes additional management parts comprising miscellaneous functions such as compiler control, HIR management, and symbol management. Application code is translated to HIR and LIR representation formats and binary code is generated for the target machine by means of COINS.

- Examples of DFG extraction from scientific applications

In scientific areas such as quantum chemistry, materials science, medicine, environmental issues, and etc. complex numerical simulations and analysis are required that are impossible without employing powerful computers. As aforementioned, one approach to gain a high performance is to reducing the overall execution time through implementing critical sections on dedicated hardware accelerators.

There are many applications which are the target of high-performance computing systems. An example is a molecular orbital calculation (ERI [15]) which has a key role in theoretical prediction of various chemical properties. The amount of floating-point operations for each calculation is between 122 and 1237 and the critical path length is from 13 to 76. To execute this application efficiently, PC cluster computers or vector type computers have been utilized so far [13]. In this research, several applications are attempted as benchmark scientific applications including: one-dimensional heat and vibration equations [16], two-dimensional Poisson equation [16], and recursion calculation part of electron repulsion integral (ERI) as a quantum chemistry application, Harmonic oscillation simulation [7], Coulomb equation [17], two-dimensional heat equations and finite-difference time domain equation (fdtf). All calculations consist of ADD, SUB, and MUL operations.

**Heat, Vibration, Poisson:** Generally, two-dimensional second order partial differential equations with constant coefficients are categorized to three types: heat or diffusion equation, vibration equation and Poisson equation. Each equation has following canonical form, respectively:

\[
\frac{\partial T(x,t)}{\partial t} = A \frac{\partial^2 T(x,t)}{\partial x^2} \tag{1}
\]

\[
\frac{\partial^2 V(x,t)}{\partial t^2} = A \frac{\partial^2 V(x,t)}{\partial x^2} \tag{2}
\]
\[
\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = f(x, y)
\]  

(3)

These equations are solved by finite difference method using following expressions [16]:

\[
T(x_i, t_{j+1}) = D * T(x_i, t_j) + B * [T(x_{i-1}, t_j) + T(x_{i+1}, t_j)]
\]  

(4)

\[
V(x_i, t_{j+1}) = D * V(x_i, t_j) + B * [V(x_{i-1}, t_j) + V(x_{i+1}, t_j)] + C * V'(x_i, t_{j-1})
\]  

(5)

\[
u^{(n+1)}(x_i, y_j) = (1 - \omega) * u^{(n)}(x_i, y_j) + \frac{\omega}{4} [u^{(n)}(x_{i-1}, y_j) + u^{(n)}(x_{i+1}, y_j) + u^{(n)}(x_i, y_{j-1}) + u^{(n)}(x_i, y_{j+1}) - h^2 f(x_i, y_j)]
\]  

(6)

where, D, B, C and \( \omega \) are constants. By using Eq. 6 which is an iterative equation (referred as successive over relaxation method), final \( u(x, y) \) is calculated as a converged form. In the next stage, benchmark DFGs are manually extracted from Eq. 4, 5 and 6, then mapped on the LSRDP by utilizing a mapping tool which will be explained in the following section. However, it is inefficient to map only small DFGs which are extracted directly from Eq. 4-6. Therefore, larger DFGs are produced through combining the smaller ones together.

![Data flow graph of basic heat equation](image1)

![Combined data flow graph of Heat equation](image2)

For example, in Heat equation, the extracted DFG corresponding to Eq. 4 can be shown as Fig. 4. This finite difference equation shows that the next point during the time evolution process \((x_i, t_{j+1})\) is obtained by using current three points: \((x_i, t_{j+1})\), \((x_{i-1}, t_j)\) and \((x_{i+1}, t_j)\). This equation can be applied to a one-dimensional \(N+1\) points: \(x_{i-N/2} \sim x_{i+N/2}\) and it can be repeated \(M\) times to calculate from \(t_j\) to \(t_{j+M}\). By extending that
equation over the space and time dimensions, the final computation structure will correspond to the DFG in Fig. 5.

By combining \( N \) DFGs over the space and \( M \) over the time directions, finally we obtain a larger DFG with \( N \) inputs and \( N-2^M \) outputs as well as \( 4^N(2N-2^M)M/2 \) operations. Consequently, by implementing the obtained DFG on the LSRDP, numerous operations are possible to be executed in each local clock of the LSRDP pipelined architecture. Similar to above mentioned DFG generation procedure is applicable to the basic DFGs of vibration and Poisson equations.

Fig. 6. Electron Repulsion integral recursion equations

ERI: Computation of molecular integrals is a problem of paramount importance in theoretical studies of molecular systems. As the system gets larger, computation of electron repulsion integrals becomes one of the most laborious and time-consuming steps in the process.

Recursion calculation equations [15] are shown in Fig. 6. In these equations, the terms \((p,s,ss)^{(0)}\) and \((p,p,p,p)^{(0)}\) are objective integral values. \((ss,ss)^{(0)}\) is an initial integral value, and all the other values are coefficients. Since indices \(i, j, k, l\) correspond to three space components \(x, y, z\), therefore each integral expression has multiple components. For example, \((p,p,p,p)^{(0)}\) has 81 components and corresponding DFG has 81 outputs. Dividing DFG to smaller ones would be useful if the DFG size is larger than the number of resources in the LSRDP. In this case, vertical partitioning of DFG is one solution to decrease the number of outputs and operations. Data flow graphs extracted semi-automatically from the target scientific applications include Heat, Vibration (Vib), Poisson (Poi) and ERI.

One of the requirements for gaining a high performance is to running DFGs which have smaller byte per flop (B/F) requirement. This means that the application is process-intensive rather than memory-intensive which is indicating a large number of processes versus smaller number of memory transactions. Therefore, another group of applications were studied in this project which are featuring smaller B/F or deep calculation dependency requirement compared with the above applications. A brief review on these applications is given below.

Phase space simulation (Harmonic Oscillator) using Runge-Kutta simulation:

Following equations are representing pure harmonic oscillator function [7]. Runge-Kutta simulation [16] can be used for solving these equations. Eq. 10 shows a number of equations which are obtained for Runge-Kutta simulation. These equations can be described by the DFG (referred as Oscillator) in Fig. 8, which has 3/2 inputs/outputs, 38 operations and the B/F ration for this DFG is 0.53. This is the smallest DFG extracted from the application, however an expanded DFG can be generated if more than one iteration over the time space is considered. For example, in a multi-step calculation when the DFG is expanded to \( n \) step including \( N \) cascaded DFGs of the smallest one (for the sake of implementing a deep DFG rather than a shallow one), the number of operations will be \( N \times 38 \), while the number of inputs/outputs does not change, therefore, B/F ratio will become \( N \) times smaller than that of the smallest DFG.
\[ H = \frac{p^2}{2m} + \frac{1}{2}kq^2 \Rightarrow \frac{p^2}{2} + \frac{q^2}{2}, (m = k = 1) \]  

(7)

\[ \frac{dq}{dt} = \frac{\partial H}{\partial p} = p \]  

(8)

\[ \frac{dp}{dt} = -\frac{\partial H}{\partial q} = -q \]  

(9)

\[ k_1^{(q)} = y_0^{(p)}, k_1^{(p)} = -y_0^{(q)} \]

\[ k_2^{(q)} = y_0^{(q)} + h\frac{k_1^{(q)}}{2}, k_2^{(p)} = y_0^{(p)} + h\frac{k_1^{(p)}}{2} \]

\[ k_2^{(q)} + = k_2^{(p)}, k_2^{(p)} + = -k_2^{(q)} \]

\[ k_3^{(q)} = y_0^{(q)} + h\frac{k_2^{(q)}}{2}, k_3^{(p)} = y_0^{(p)} + h\frac{k_2^{(p)}}{2} \]

\[ k_3^{(q)} + = k_3^{(p)}, k_3^{(p)} + = -k_3^{(q)} \]

\[ k_4^{(q)} = y_0^{(q)} + hk_3^{(q)}, k_4^{(p)} = y_0^{(p)} + hk_3^{(p)} \]

\[ k_4^{(q)} + = k_4^{(p)}, k_4^{(p)} + = -k_4^{(q)} \]

\[ y_1^{(q)} = y_0^{(q)} + (k_1^{(q)} + 2k_2^{(q)} + 2k_3^{(q)} + k_4^{(q)}) / 6, \]

\[ y_1^{(p)} = y_0^{(p)} + (k_1^{(p)} + 2k_2^{(p)} + 2k_3^{(p)} + k_4^{(p)}) / 6 \]  

(10)
Coulomb equation: Another attempted application is the Runge-Kutta simulation for solving Coulomb equation [17] (Eq. 11). Coupled Runge-Kutta equations (Eq. 12 and 13) indicating second order differential calculations are achieved for solving Coulomb equation.

\[
\frac{d^2y}{dx^2} + \left( \pm 1 + \frac{2z}{x} - \frac{l(l+1)}{x^2} \right) y = 0
\]  

(11)
\[
\frac{dy^{(1)}}{dx} = y^{(2)}
\]  

(12)
\[
\frac{dy^{(2)}}{dx} = \left( \pm 1 + \frac{2z}{x} - \frac{l(l+1)}{x^2} \right) y^{(1)}
\]  

(13)

2.3. Interface to the LSRDP

Dedicated instructions for managing transactions between GPP and the accelerator ought to be defined and used in the code development targeting the SFQ-LSRDP. GPP should interact with LSRDP during executions by managing control signals to transfer required input/output data from/to memory to/from the stream buffers, transmitting configuration data from memory to the LSRDP for reconfiguration, invoking the LSRDP to start operating, synchronizing between LSRDP and GPP and so on. Table 1 describes the list of instructions and functionality of each one briefly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf_LSRDP</td>
<td>configures the LSRDP’s data-path</td>
</tr>
<tr>
<td>set_io_info</td>
<td>sets the data to be read as inputs of the LSRDP for several iterations in the case of stream buffers.</td>
</tr>
<tr>
<td>run_LSRDP</td>
<td>runs the LSRDP according to the configured information and the input data. This is a non-blocking call.</td>
</tr>
<tr>
<td>sync_LSRDP</td>
<td>waits for the RDP to finish its task</td>
</tr>
<tr>
<td>load_to_SPM</td>
<td>loads data to be processed from the main memory to the SPM</td>
</tr>
<tr>
<td>store_from_SPM</td>
<td>stores computed data from the SPM to the main memory</td>
</tr>
<tr>
<td>set_IO_data_seq</td>
<td>sets the memory pointer to the area that LSRDP should read from</td>
</tr>
</tbody>
</table>

To support the LSRDP within application code two memory configurations are assumed. In the first configuration, a main memory and input/output buffers are the main components of memory system. Input/output data might be rearranged depending on target application demands. New instructions which are defined to support the LSRDP are conf_LSRDP, set_IO_info, run_LSRDP and sync. In the second configurations, it is assumed that a scratchpad memory is used as well. Instructions to support the LSRDP...
are conf_LSRDP, set_IO_info, load_to_SPM, store_from_SPM, run_LSRDP and sync. 0 displays a typical modified code for both above cases, when there is no scratchpad memory and in presence of scratchpad memory.

### 2.4. Mapping tool

During the mapping process, first the DFG nodes are located on appropriate positions (PEs) on the LSRDP. This is similar to the well-known placement problem [18]. Generally, minimizing the total connection length or the maximum connection length are objective goals, whereas in the LSRDP the main goal is to minimize the maximum connection length that directly affects the ORN sizes. Connection length can be calculated based on euclidean or manhattan distances. We use the latter one within the design process.

Routing process is the next stage that makes routes between the PEs by means of ORNs and transfer units [11]. It is supposed that each PE include a transfer unit for transferring data as well as a FU for implementing operations. For each connection it is aimed to find a shortest path between the source and destination nodes. ORNs provide connections between two consequent rows and therefore the connection length between two PEs in consequent rows should be less than the available connection resource size. Fig. 10 displays the flow of mapping process.

#### 2.4.1. Placement

Placing DFG nodes is performed in three steps including the input nodes placement, operations placement and output nodes placement. Input/output nodes of the DFG should be located on appropriate input/output ports of the LSRDP on top/down boundries. ORNs as routing resources exist between the first/last LSRDP rows and input/output ports. The main objective is to reduce the connection length between input/output ports and PEs in the first/last row of the LSRDP. Since DFG nodes are placed based on the location of their parents inside the LSRDP, placing input nodes is performed in a different manner from the placing output ports. Proper locations for output nodes can be determined based on the position of parent nodes which have already been placed.

In [11] a simple placement algorithm for input nodes referred as fan-out based placement (See Appendix A), I/O nodes are placed with respect to their fan-out or the total number of children. First, input nodes of DFG are prioritized with respect to their fan-out numbers, and then the placement algorithm looks for proper locations for them over the input ports to minimize the longest connection length. We have also proposed a different algorithm referred as proximity factor-based placement for input nodes (See Appendix B). The main intuition behind the proposed algorithm is to locate input nodes which have stronger connections to each other in a closer distance. Proximity factor is representing the strength of forces that input ports can exert to each other depending upon the connectivity of their descendants in sub-trees. This algorithm is extended to locate DFG nodes on the PEs as well.

For locating DFG operations on the PEs we propose a naive algorithm which tries to find a suitable position for each with respect to the positions of its parents. Obviously, due to availability of only unidirectional routing resources from each row to its consequent row, each node should be placed in the lower rows where the parents are in upper rows of array. This process is run after the input nodes placement, therefore starting from the nodes with the lowest ASAP level, their position are determined, afterward this process is carried on for the next levels. To place each node, first an initial row is determined, which is the row next to the lowest row wherein a parent of intended node is located. Starting with the initial row, every unoccupied PE is examined in terms of the connection length criterion. For each PE which is being attempted, the following term is calculated as Eq. 14.

\[
TCL = \sum_{i=1}^{n} \left[ \frac{d_{h}}{d_{v}} \right]
\]  

where, \(d_{h}/d_{v}\) is the horizontal/vertical distance of PE assigned to \(i\)-th parent of the node to the PE which is being examined, and \(n\) is the number of parents. After calculating TCL for every PE in the initial row and its succeeding rows, the PE with minimum amount of TCL is chosen as the place for locating the DFG node. If there would be more than one PE with the minimum TCL, among them a PE with minimum total horizontal distances to its parents will be chosen.
2.4.2. Routing

Routing process searches a route between source and target PEs in the LSRDP in two steps. First, each net connecting a pair of PEs is globally routed using the available resources including ORNs and TUs. Afterward, a micro-routing algorithm finds paths through the cross-bar switches for the nets connecting ORN's inputs to outputs.

Routing global nets:
The input of routing algorithm is a netlist representing a list of interconnections (nets) between DFG nodes as well as placement information including the position of nodes in the LSRDP array. According to the underlying LSRDP architecture, graph model for routing can be presented as Fig. 11 in which the vertices and edges are representing the PEs and interconnection resources, respectively. The edges are two-terminal and unidirectional. We suppose that $G=(V,E)$ is the graph to describe the LSRDP routing layout. $(s, d)$ is a global net which should be routed between source ($v_s$) and destination ($v_d$) vertices in the graph. A path $P_{s,d}$ is constructed through the routing algorithm, and consists of a sequence of vertices represented as $P_{s,d}=(v_s=v_s^0,v_s^1,...,v_s^{D_s(v_s,v_d)-1},v_d=v_d^{D_s(v_s,v_d)})$, while $D_s(v_s,v_d)$ is the vertical distance between source and destination vertices and $v_s^i$ ($i \in [0,...,D_s(v_s,v_d)]$) is an intermediate vertex corresponding to the $i$-th hop of the path. Obviously, vertical distance of two consecutive vertices on the path (e.g. $v_s^i,v_s^{i+1}$) is equal to one. $D_h(v_1,v_2)$ is defined as the horizontal distance of two vertices $v_1$ and $v_2$. A capacity is defined for every edge and vertex as well. It is initiated to 1 for all edges, however the capacity of vertices varies from 1 to 3 based on the number of available transfer units at the corresponding PE (each vertex in the graph is associated with a PE in the LSRDP array within placement process). As aforementioned, every PE can implement from 1 (when FU is used for implementing an operation) up to 3 TUs (in case of availability of both FU and TU). The initial capacities of vertices are assigned after the placement, and alter during the routing process. Once a net is routed, the capacity of involved vertices and edges decreases by 1.
For each net \((s, d)\), a target connection length is defined as \(D_{Ts,d} = D_h(v_s,v_d)/D_s(v_s,v_{d})\). The objective of the proposed routing algorithm is to minimize the deviation of horizontal connection lengths from \(D_{Ts,d}\):

\[
\text{Minimize } \sum_{i=0}^{D_h(v_s,v_d)} \left| D_{Ts,d} - D_h(v_{s+i},v_{d+i}) \right|
\]

Two algorithms have been proposed for global routing. The first routing algorithm is an iterative procedure of finding a path between source and destination PEs. All DFG edges are dealt with as two-terminal nets even if some of them have common source or destinations. A modified maze router runs so that several paths start at the source, and are expanded until one of them reaches the target. Afterward, all the employed resources on the path are labeled as used ones which are no longer available for the next routes. Therefore, a higher priority is given to the critical nets which are located on DFG’s critical paths.

A main difference with the traditional maze routers [18] is that ours tries to make the horizontal connection length closer to \(D_T\) at each step rather than minimizing the connection lengths. The time complexity of this algorithm is \(O(MCL^3)\), which is indicating an exponential growth, hence a significant execution time for the long nets.

The second routing algorithm which is referred as quick router addresses the long execution time that the first algorithm incurs. Starting from the source, for each PE satisfying the MCL size constraint, summation of horizontal distances to source and destination PEs is calculated and a PE with minimum value is chosen as an intermediate node and it will be considered as a source PE for the next iteration as well. This is repeated for the consequent rows until reaching to the destination. When a route fails due to lack of transfer node or violating MCL constraint, the algorithm backtracks to an upper row by choosing another PE which has the smallest connection lengths to source and destination. This algorithm falls in the category of branch and bound algorithms while the former one tries all possible routes exhaustively. The latter algorithm demonstrates a very short routing time in the range of seconds for the longest routes. Fig. 12 displays how two algorithms work for a given net connecting a source PE and destination PE. In the first algorithm all possible paths are searched exhaustively to find the best route meeting the constraints as well as satisfying the abovementioned goal. Two of possible route have been indicated in the figure. Second algorithm only tries the best possible connection among several ones in each step and continues till reaching to destination. Other routes are tried when one fails.

Fig. 13 shows examples of mapping result generated for Heat-16x2 and Vib-8x2 DFGs (a postscript file is generated by the mapping tool). Red lines represent the connections corresponding to MCL.

![Routing graph model for the LSRDP when MCL=2](image)
2.4.3. Micro-routing algorithm for the ORN

The aim of micro-routing algorithm is to route the nets through the CBs inside ORN. For each net passing through the ORN (referred as micro-net), CBs are configured to create a path from one input to one or more outputs of the ORN.

Here are some definitions:

- $ORN^k$: $k$-th ORN located between LSRDP rows $i$ and $i+1$.
- $\{i_1^k, i_2^k, ..., i_{2^k-1}^k\}$: (indexes of) the inputs of $ORN^k$.
- $\{o_1^k, o_2^k, ..., o_{2^k-1}^k\}$: (indexes of) the outputs of $ORN^k$.
- $c_b^k_{i,inp_1}, c_b^k_{i,inp_2}$: inputs of the CB located in row $i$ and column $j$ of the $ORN^k$.
- $c_b^k_{i,out_1}, c_b^k_{i,out_2}$: outputs of the CB located in row $i$ and column $j$ of the $ORN^k$.

The inputs to this algorithm are as follows:

- LSRDP array dimensions ($W$ and $H$).
- ORN dimensions including the number of CB rows and the number of CBs in each row (i.e. $M$, $N$).
- For each $ORN^k$, $k=1..H-1$, list of micro-nets: $\{(i_1^k, o_1^k) \mid j \in [2, ..., 2^k-1]\}$, the micro-net connecting $i_1^k$ to $o_1^k$.

The output of routing algorithm is the list of paths so that each path includes a list of CBs as well as configuration of each CB located on the path.

We propose a naïve algorithm for solving the above routing problem (Fig. 15). Firstly, for each ORN, the outputs of CBs located in the last column are being labeled with the index of the inputs of corresponding nets. Unlabeled CB outputs are initiated with -1. Through a back-tracking algorithm, inputs of CBs in each column are labeled with respect to the CBs’ output labels until reaching to the first column. A path will be easily recognized by grouping CBs with similar labels. The time complexity of this algorithm is $O(M \times N)$. Fig. 14 shows a result of routing on a piece of the ORN located between 3rd and 4th rows of the LSRDP for a DFG extracted from Heat-8x2 [16] application. The micro-nets should be routed are: $\{(I_{13}, O_{14}), (I_{17}, O_{18}), (I_{18}, O_{17}), (I_{18}, O'_{29}), (I_{20}, O'_{20}), (I_{24}, O'_{22}), (I_{24}, O'_{30}), (I_{25}, O'_{26}), (I_{31}, O'_{33}), (I_{32}, O'_{32})\}$. For example, according to the netlist, input 18 of ORN should be connected to outputs 17 and 24. The
path starting from input 18 splits at the middle; therefore two separate paths reach to the outputs 17 and 24 from the branch in the middle.

It can be proved that using the proposed ORN architecture, every output of the ORN is reachable from any input located in the horizontal distance up to \textit{MCL}. It should be noted that the nets’ congestion for ORNs varies from upper rows of the LSRDP, where ORNs are highly congested to the lower rows, where they are usually sparse.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure14}
\caption{An example of ORN micro-routing for Heat 8x2}
\end{figure}

**ORN Micro-routing algorithm:**

for each \(k=1\) to \(H-1\) (routing micro-nets for each ORN)

for every net \((i^k_j, o^k_j)\) of \(ORN^k\)

\[ CB^k_{O_{[j/2]}^k} \setminus \{Out_{i^k_j} - O_{i^k_j}^a\} = i^k_j /\text{output of the CB with the output } O_{i^k_j}^a \text{ is being labeled with } i^k_j. \]

for each net \((i^k_j, o^k_j)\), following steps on \(ORN^k\) are repeated

for each column \(c\) of \(ORN^k\), \(c = N\) to 1

\(//\) back-tracking from the last column

for \(r\)-th CB in column \(c\) (i.e. \(CB^k_{r,c}\)), \(r=1\) to \(M\)

if \[ CB^k_{r,c}(\text{out}_1) < CB^k_{r,c}(\text{out}_2) \]

\[ CB^k_{r,c}(\text{inp}_1) = CB^k_{r,c}(\text{out}_1), CB^k_{r,c}(\text{inp}_2) = CB^k_{r,c}(\text{out}_2) \]

else

\[ CB^k_{r,c}(\text{inp}_1) = CB^k_{r,c}(\text{out}_2), CB^k_{r,c}(\text{inp}_2) = CB^k_{r,c}(\text{out}_1) \]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure15}
\caption{Micro-routing algorithm for ORN}
\end{figure}

\subsection{Connection Length Measurement}

As aforementioned, in the basic placement algorithm the goal is to decrease MCL size. Also, the connection length between source and destination nodes is evaluated as the horizontal distance of two nodes (connection length= \(h=d_h\) : horizontal distance). For two nodes located in two consecutive rows \((d_i=1)\), the only possible way for routing is through the available ORN resources. On the other hand for two nodes placed in inconsecutive rows (vertical distance: \(d_v>1\)), it is possible to use intermediate TUs for routing. In this way, the connection can be segmented to \(d_v-1\) connections between consecutive rows.

Considering above point, we use an alternative measurement for the connection length as \(l_{hv}=\frac{d_h}{d_v}\). In Fig. 16 (left-side) the two definitions of connection length have been displayed. In right-side two connections (denoted by 1 and 2) can be seen so that \(d_{h1}=d_{h2}=3\), while they have different vertical distances \((d_{v1}=1, d_{v2}=3)\). For connection 1, connection length would be 3 and the only possible way for routing from
src to dest1 is via the ORN switches. On the contrary, for connection 2, it is possible to use available intermediate TUs and break it into three segmented connections from src to dest2. This results in reducing connection length to one.

By using above new definition for connection length measurement, we modified the placement algorithm such that the vertical connection length becomes effective in calculating the cost function. Fig. 17 shows an example of placing a node which has two parents (P1 and P2) already placed on two PEs. Obviously due to the routing resource constraints (particularly of being unidirectional) descendant nodes can be placed on the PEs of their parents’ succeeding rows (indicated as candidate rows in Fig. 17). The connection length has been calculated based on two abovementioned measurements for each candidate PE. By using \( l_h \) for connection length measurement, PE2 is the best choice which gives the minimum MCL equal to 2. In the second candidate row, choosing PE8 gives the same result, however PE2 is proffered due to smaller vertical distance. On the other hand, connection length measurement based on \( l_{hv} \) cause to choosing PE2 which gives the least value of MCL equal to 1 in the first row and PE6, PE7 and PE8 in the second row have the same result, but due to smaller vertical distance, PE2 is eventually chosen.

2.4.5. Integrated Placement and Routing

In the modified flow introduced in Section 6.4.1 it is assumed that placement process is completely accomplished and then the routing process is started. In this case, there might be no further chance for changing the position of already placed nodes and use the associated PEs for routing critical nets. In an attempt to increase the routability of nodes, we have proposed an integrated placement and routing procedure. The difference with previous flow is in being stepwise placement and routing for each node. In other words, each node of DFG is placed onto the LSRDP and then related connections are routed to its parents (which have been already placed). By using this approach further improvement in MCL reduction might be achieved.
Acknowledgement

This research was supported in part by Core Research for Evolutional Science and Technology (CREST) of Japan Science and Technology Corporation (JST).

References


Appendix A: Fan-out Based Input Nodes Placement

This approach is similar to the placement problem so that it is locating input/output nodes of the DFG on appropriate positions around LSRDP. In the LSRDP it is assumed that input and output ports are located in the top and bottom borders, respectively. Between input/output ports and PEs in the first/last rows ORNs are located to establish connections related to input/output ports. In this step, the main objective is to reducing the connection length between the ports and PEs. Port positioning is performed in four steps:
- placing input ports on the top boundary of RDP
- routing nets from input ports to their children in the first row of LSRDP
- placing output ports on the bottom boundary of RDP
- routing nets from parent nodes to the output ports

We show that how this problem can be solved for input ports. A similar approach can be used for output ports positioning. Here there are some assumptions:
- \( n_i \): the number of children of input node \( i \)
- \( C_1, C_2, C_3, \ldots, C_{n_i} \): location (column number) of children \( i_1, i_2, \ldots, i_{n_i} \).
- \( X \): location of the input node \( i \)

The total connection length is summation of distance of the input node to the children in horizontal direction: 
\[
TCL = |C_1-X| + |C_2-X| + \ldots + |C_{n_i}-X|
\]
and the objective function is to minimizing TCL. Depending on the number of children, the optimal location for the input port can be determined as follows (Fig. 19):
- \( n_i = 1 \) : \( X = C_1 \)
- \( n_i = 2 \) : \( C_1 \leq X \leq C_2 \)
- \( n_i = 3 \) : \( X = C_2 \)
- \( n_i \geq 2 \) : \( X = C_j, j=2\ldots n_i-1 \)

![Fig. 19. Input ports positioning](image-url)
Appendix B: Proximity-Factor Based Input Nodes Placement

Placing input nodes considering only the location of their descendants might result in large MCLs as shown in Fig. 20. In a piece of mapping result displayed in Fig. 20, input nodes 0 and 24 are located in a close distance to their direct children (located in the first row) however they have a long distance to each other and to a common descendant node which is positioned in the third row as well. This kind of placement without paying attention to the location of descendants make some long connections which affect the MCL size. One solution to cope with this issue is to locate input nodes which have strong connections to each other in a closer distance. We define a factor referred as proximity factor to represent the strength of forces that PEs can exert to each other.

**Definition- Proximity factor:** for each pair of intout nodes $i$ and $j$ their proximity factor is calculated as:

$$p_{i,j} = \sum_{k \in S_{i,j}} \frac{1}{D_{k,i}}$$

while $D_{k,i}$ is the distance of common descendant node $k$ to the input ports $i$ (and $j$). The distance of a node to its related input port can be calculated as its ASAP (As soon as possible [5]) level of execution. $S_{i,j}$ is the set of common descendants for the input nodes $i$ and $j$. Larger number of common descendants with smaller distance to the parents pair result in larger proximity factor and therefore, corresponding nodes should be placed in a closer distance to each other. Proximity factor for different pairs of the sample DFG in Fig. 21 can be calculated as follows:

$$S_{1,2} = S_{2,1} = \{4,6,7\}, p_{1,2} = p_{2,1} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = 1.08$$

$$S_{1,3} = S_{3,1} = \{7\}, p_{1,3} = p_{3,1} = 0.25,$$

$$S_{2,3} = S_{3,2} = \{7\}, p_{2,3} = p_{3,2} = 0.25$$

Obviously, $S_{1,1} = S_{2,2} = S_{3,3} = \infty$.

**Input nodes placement algorithm:** A heuristic algorithm is introduced below which employs the proximity factor for positioning input nodes of a DFG onto the LSRDP input ports.

**Definitions:**

$L$: is the input ports array which stores the list of placed nodes such that the indexes indicate the location of corresponding input node.

$l$ and $r$: denote the index of candidate locations in $L$ for placing the under process input node.

$C_{l,m}$ and $C_{r,m}$: show the amount of proximity of an under process node $m$ to the previously placed input nodes which have been located between $l$ and $r$ in array $L$. The node $m$ will be placed in location $l$ if $C_{l,m}$ is larger than $C_{r,m}$ otherwise, it will be located in location $r$. $C_{l,m}$ and $C_{r,m}$ are calculated as:

$$C_{l,m} = \sum_{i=l+1}^{c} p_{i,m} \times \frac{1}{|l-i|}$$

$$C_{r,m} = \sum_{i=r+1}^{c} p_{i,m} \times \frac{1}{|r-i|}$$

In Eq. 4 and 5, $p_{i,j}$ is the proximity factor of node $m$ and node $i$ that has already been placed. The second term is the inverse of distance of candidate location ($l$ or $r$) to the location of node $i$. In this way, the strength of proximity to already placed nodes is examined and one of the candidate locations ($l$ or $r$) is chosen.

**Placement alg.**

1. Construct matrix $P$ including the proximity factors for each pair of input nodes ($n$ is the number of input nodes). Initialize $L = \Phi$.
2. Find node $m$ with the highest proximity factor from the first row of matrix $P(i=1)$ and place it in array $L$ so that $L[n/2] = m$.

Initialize $l$ and $r$ to $n/2 - 1$ and $n/2 + 1$, respectively.

3. Find the next node ($m$) with the highest proximity factor from the first row of matrix $P(i=1)$.

4. Calculate $C_{l,m}$ and $C_{r,m}$ using Eq. 4 and Eq. 5.

5. if $C_{l,m} > C_{r,m}$:
   - $l = l + 1$, $L[l] = m$ (node $m$ is placed in the location $l$)
   else:
   - $r = r + 1$, $L[r] = m$ (node $m$ is placed in the location $r$)

6. if still there is unplaced input node, go to step 4.

Fig. 20. A piece of mapping result for Poisson-3x3 DFG

Fig. 21. A sample DFG
Appendix C: How to Use Tool Chain in Practice (A Brief Tutorial)

As an application is divided into software and hardware parts, therefore the developed tool chain can generate two main outputs: an assembly code for the GPP and a configuration file for the LSRDP. The DFGs are extracted from the application code and are fed as inputs to the mapping tool. Fig. 22 displays description format for a DFG. Mapping tool generates the configuration file after performing placement, routing and micro-routing of the DFGs on the LSRDP. As an example we consider the Heat application and its corresponding C code. Fig. 23 and Fig. 24 show the c code and generated DFGs described in our developed format.

```c
#define MAIN
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

RDP_LI calc (RDP_LI nLine, RDP_LI nGlue, RDP_LI nSlide, RDP_LI *SB_IN, RDP_LI *SB_OUT)
{
    RDP_LI i, j, k, iL;
    RDP_LD xb[2][WIDTH * 2];
    RDP_LI nOut = WIDTH - nGlue * 2;
    RDP_LI nStart = WIDTH - nGlue * 2;
    xb[1][0] = 0;
    xb[1][1] = 0;
    for (i = 0; i < nGlue; i++) {
        SB_BUF[i] = SB_IN[i];
    }
    for (iL = 0; iL < nLine; iL++) {
        for (i = 0; i < nOut; i++) {
            SB_BUF[nGlue + WIDTH * iL + i] = SB_IN[nOut * nLine + i];
        }
        for (i = 0; i < nGlue; i++) {
            SB_BUF[nGlue + WIDTH * nLine + i - nGlue * 2] = SB_IN[nGlue + nOut * i + i];
        }
    }
    return 0;
}

RDP_LI main() {
    RDP_LI k_curr, nLine, nSlide, nGlue;
    RDP_LD XX[2][POINTS_W * 10];
    RDP_LD *pXX_in, *pXX_out, *ptmp;
    pXX_in = INIDAT;
    pXX_out = XX[1];
    cal_params_slide(&nSlide, &nLine, &nGlue);
    k_curr = 0;
    for (; k_curr < NVSTEP; k_curr++) {
        add_gpp_part(nGlue, nLine, pXX_in, pXX_out);
        ptmp = pXX_in;
        pXX_in = pXX_out;
        pXX_out = ptmp;
    }
    return 0;
}
```

---

Fig. 22. DFG description format

---

Fig. 23. C code for Heat
Fig. 24. DFG (Heat 8x1) generated from the original C code

Generating configuration file for the LSRDP: Following command (command 1) is used for running mapping tool. The inputs are the architecture description file and DFG file. The output includes a mapping result representing a configuration file as well as several report files. Mapping result is also generated as postscript file for schematic view and can be displayed by a '.ps' file viewer. Fig. 25 shows a mapping result for DFG Heat-8x1. Red lines denote the connections with maximum length.

command 1:  /rdp-mapping.exe  lsrdp-arch.dat  dfg-file.txt

Also, Fig. 26 display a sample generated report for Heat-8x1 in four sections. First section gives information on the input DFG. The second section displays the information of the generated map including the occupied PEs and the implemented operations by each one of them. The third section gives information on the utilization rate of the PEs and its components i.e. FUs and TUs, as well as connections, MCL and total estimated area in terms of the number of Josephson Junctions. Last section of the report describes general and detailed information on the connections, source and destination PEs, their lengths and etc.

Generating executable code for GPP: In the next step after generating configuration file for the LSRDP, an executable file should be generated for the GPP as well. LSRPD-specific instructions (already introduced in Table 2) are inserted by the programmer in the original C code, then it is converted to an assembly code by means of COINS [3]. A header file for declaring LSRDP-specific instructions is shown in Fig. 27. Fig. 28 shows a part of modified C code of Heat application. Command 2 in a COINS command for compiling a C Code to a MIPS-based assembly code. Generated assembly code for Heat-8x1 is partially shown in Fig. 29.

command 2: java  coins.driver.Driver  -S  file-name.c
Section 1: Input DFG info:
Total_No_of_Nodes_in_DFG = 34
DFG_inputs_no = 14
DFG_outputs_no = 4
NoOfGeneralInputs(NV) = 6
NoOfInputsWithSimilarImmValue(D) = 4
NoOfInputsWithSimilarImmValue(B) = 4
No_Of_Operations_Are_Mapped_On_RDP = 16

Section 2: RDP Map:
FU_5: Row=0, Col=5, Node= 14, Operation= MUL
FU_6: Row=0, Col=6, Node= 7, Operation= ADD
FU_7: Row=0, Col=7, Transfer & Node= 6, Operation= MUL
FU_8: Row=0, Col=8, Transfer & Node= 15, Operation= ADD
FU_9: Row=0, Col=9, Transfer & Node= 22, Operation= MUL
FU_10: Row=0, Col=10, Transfer & Transfer
FU_21: Row=1, Col=5, Transfer & Node= 8, Operation= MUL
FU_23: Row=1, Col=7, Transfer & Node= 16, Operation= MUL
FU_24: Row=1, Col=8, Node= 23, Operation= ADD
FU_25: Row=1, Col=9, Transfer & Node= 30, Operation= MUL
FU_26: Row=1, Col=10, Node= 31, Operation= ADD
FU_37: Row=2, Col=5, Transfer
FU_38: Row=2, Col=6, Node= 9, Operation= ADD
FU_39: Row=2, Col=7, Transfer & Node= 24, Operation= MUL
FU_40: Row=2, Col=8, Transfer
FU_41: Row=2, Col=9, Transfer & Node= 32, Operation= MUL
FU_53: Row=3, Col=5, Transfer
FU_54: Row=3, Col=6, Node= 17, Operation= ADD
FU_56: Row=3, Col=8, Node= 25, Operation= ADD
FU_58: Row=3, Col=10, Node= 33, Operation= ADD
FU_69: Row=4, Col=5, Transfer
FU_70: Row=4, Col=6, Transfer
FU_71: Row=4, Col=7, Transfer
FU_73: Row=4, Col=9, Transfer
FU_85: Row=5, Col=5, Transfer
FU_86: Row=5, Col=6, Transfer
FU_87: Row=5, Col=7, Transfer
FU_89: Row=5, Col=9, Transfer
FU_101: Row=6, Col=5, Transfer
FU_102: Row=6, Col=6, Transfer
FU_103: Row=6, Col=7, Transfer
FU_104: Row=6, Col=8, Transfer
FU_105: Row=6, Col=9, Transfer
FU_106: Row=6, Col=10, Transfer

Section 3: Statistics on the PE utilizations, connections, MACL and Area
RDP_max_width= 6
RDP_max_depth= 4
no_of_occupied_FUs= 68
no_of_pure_FUs= 16
no_of_transfer_FUs= 61
Fraction_of_FU= 12 %
Fraction_of_T= 75 %
Fraction_of_FUT= 12 %
Fraction_of_FUT= 1 %
no_of_fully_occupied_PEs= 0
No_Of_Operations_In_RDP_Rows: MUL= 3 ADD= 2 MUL= 3 ADD= 2 ADD= 1 MUL= 2 ADD= 3
max_fo_for_input_ports: 2
node_w_max_fo: node: 6, Op= MUL
Average_fan_out= 1.06
MCL:
In_Row 0= 2 (SrcFU= 8 DestFU= 10 (Row= 0,Col= 8))
WxH=6 x 4, MCL= 2, PEType= 2, NoOfTUsInPE= 1
LSRDP area: ORNs= 221.53KJJ, PEs= 1152.00KJJ, Total= 1373.54KJJ

Section 4: Connections
the no. of connections with the length of 0= 60
the no. of connections with the length of 1= 28
the no. of connections with the length of 2= 2
the no. of connections with the length of 3= 0
the no. of connections with the length of 4= 0
... Connection lengths in descending order:
conection length between TU/FU_40 and node_33/FU_58 is :2
conection length between input node 10 and TU/FU_10 is :2
conection length between input node 10 and node_22/FU_9 is :1
conection length between node_22/FU_9 and node_23/FU_24 is :1
conection length between input node 2 and node_6/FU_7 is :1
conection length between node_6/FU_7 and node_23/FU_24 is :1
conection length between input node 18 and node_15/FU_8 is :1

END of Report

Fig. 25. A sample mapping result for DFG Heat-8x1

Fig. 26. Report file generated by the mapping tool
#define MAIN
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "lsrdp_lib.h"

RDP_LI calc_LSRDP (RDP_LI nLine, RDP_LI nGlue, RDP_LI nSlide,
RDP_LD *SB_IN, RDP_LD *SB_OUT)
{
    RDP_LI i, j, k, nL,
    RDP_LD xb[2][WIDTH * 2],
    RDP_LI nOut = WIDTH - nGlue * 2;
    RDP_LI Start = WIDTH - nGlue * 2;
    xb[1][0] = 0;
    xb[1][1] = 0;
    for (i = 0; i < nGlue; i++) {
        SB_BUF[i] = SB_IN[i];
    }
    for (iL = 0; iL < nLine; iL++) {
        for (i = 0; i < nOut; i++) {
            SB_BUF[nGlue + WIDTH * iL + i] = SB_IN[nGlue + nOut * iL + i];
        }
    }
    for (i = 0; i < nGlue; i++) {
        SB_BUF[nGlue + WIDTH * nLine + i - nGlue * 2] = SB_IN[nGlue + nOut * nLine + i];
    }
    ...
    return 0;
}

RDP_LI main()
{
    RDP_LI k_curs, nLine, nSlide, nGlue;
    RDP_LD XX[2][POINTS_W * 10];
    RDP_LD *pXX_in, *pXX_out, *ptmp;
pXX_in = INIDAT;
pXX_out = XX[1];
conf_lsrdrp((void*)conf_address);
cal_params_slide( &nSlide, &nLine, &nGlue );
k_curs = 0;
for ( k_curs + NVSTEP <= POINTS_H ) {
    k_curs += NVSTEP;
    run_lsrdrp((void*)dest_address);
cal_lsrdrp( nLine, nGlue, nSlide, pXX_in, pXX_out );
    add_gpp_part( nGlue, nLine, pXX_in, pXX_out );
    ptmp = pXX_in;
pXX_in = pXX_out;
pXX_out = ptmp;
}
sync_lsrdrp();
return 0;
}
Fig. 29. A part of the assembly code generated for GPP

```assembly
_bswap_16:
  subu $sp, $56
  lw   $sp, 44($sp)
  lw   $sp, 40($sp)
  lw   $sp, 36($sp)
  move $sp, $sp
  move $8, $8
  slr  $s8, $s8, 16
  srl  $s8, $s8, 16
  sll  $s8, $s8, 16
  move $2, $8
  move $sp, $sp
  lw   $sp, 36($sp)
  addu $sp, $sp, 56
  jr   $31

main:
  subu $sp, 15464
  lw   $sp, 15452($sp)
  lw   $sp, 15448($sp)
  lw   $sp, 15444($sp)
  lw   $sp, 15440($sp)
  lw   $sp, 15436($sp)
  lw   $sp, 15432($sp)
  move $sp, $sp
  la   $10, _filename_out
  lw   4($sp)
  la   $5, $LC94
  jal  _fopen
  nop
  la   $10, fp_out
  lw   2($sp)
  la   $10, _INIDAT
  move $17, $10
  add  $10, $fp, 28
  add  $10, $fp, 24
  move $4, $16
  jal  _conf_lsrdp
  nop
  add  $4, $sp, 28
  add  $5, $sp, 24
  add  $6, $sp, 32,$19
  jal  _run_lsrdp

  nop
  lw   $4,24($sp)
  nop
  lw   $5,32($sp)
  nop
  lw   $6,28($sp)
  move $7,$17
  lw   $18,16($sp)
  jal  _calc_lsrdp
  nop
  lw   $4,32($sp)
  move $5,24($sp)
  nop
  move $6,$17
  move $7,$18
  jal  _add_gpp_part
  nop
  move $2,$17
  move $17,$18
  move $18,$2
  j   $L136

$L138:
  la   $10, _fp_out
  lw   $4,0($sp)
  nop
  lw   $6,24($sp)
  nop
  lw   $7,32($sp)
  nop
  move $5,$16
  lw   $17,16($sp)
  jal  _print_data_all
  nop
  la   $10, fp_out
  lw   $4,0($sp)
  jal  _fclose
  nop
  li   $2,0
  move $sp, $sp
  lw   $sp, 15452($sp)
  nop
  lw   $sp, 15448($sp)
  nop
  lw   $16,15440($sp)
  nop
  lw   $17,15436($sp)
  nop
  lw   $18,15432($sp)
  nop
  lw   $19,15432($sp)
  nop
  addu $sp, $sp, 15464
  jal  _conf_lsrdp
  nop
  jr   $31
```

---

27